VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA STROJNÍHO INŽENÝRSTVÍ
ÚSTAV AUTOMATIZACE A INFORMATIKY

FACULTY OF MECHANICAL ENGINEERING
INSTITUTE OF AUTOMATION AND COMPUTER SCIENCE

KAUZÁLNÍ ANALÝZA CHYB ŘÍDÍCÍCH PROGRAMŮ

CAUSAL ANALYSIS OF CONTROL PROGRAMS ERRORS

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE — HANA ČAPKOVÁ
AUTHOR

VEDOUCÍ PRÁCE — DOC. ING. BRANISLAV LACKO, CSC.
SUPERVISOR

BRNO 2013

Vysoké učení technické v Brně, Fakulta strojního inženýrství

Ústav automatizace a informatiky
Akademický rok: 2012/2013

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

student(ka): Hana Čapková

který/která studuje v **bakalářském studijním programu**

obor: **Aplikovaná informatika a řízení (3902R001)**

Ředitel ústavu Vám v souladu se zákonem č.111/1998 o vysokých školách a se Studijním a zkušebním řádem VUT v Brně určuje následující téma bakalářské práce:

Kauzální analýza chyb řídících programů

v anglickém jazyce:

Causal analysis of control programs errors

Stručná charakteristika problematiky úkolu:

Vypracovat srovnávací studii postupů pro analýzu chyb řídících programů jako příspěvek ke zlepšení kvality při tvorbě programů pro aplikace řízení v reálném čase.

Cíle bakalářské práce:

1. Zpracovat typologii chyb řídících programů
2. Analýzovat vlivy na chyby v programech
3. Provést srovnávací studii různých postupů pro nalézání příčin chyb v programech

Vedoucí bakalářské práce: doc. Ing. Branislav Lacko, CSc.

Termín odevzdání bakalářské práce je stanoven časovým plánem akademického roku 2012/2013.

V Brně, dne 26.10.2012

L.S.

_____	_____
Ing. Jan Roupec, Ph.D.	prof. RNDr. Miroslav Doupovec, CSc., dr. h. c.
Ředitel ústavu	Děkan fakulty

ABSTRAKT

Bakalářská práce se zabývá problematikou odhalování chyb v softwarových programech. Práce obsahuje hierarchicky organizovanou typologii softwarových chyb, získanou rešeršemi z dostupné literatury, analýzu vlivů na vznik softwarových chyb vedoucí k typologii příčin chyb inspirované kognitivní psychologií a srovnání technik a metodik softwarového vývoje z hlediska validace a verifikace produktu.

Bakalářská práce byla zpracována s ohledem na potřeby a požadavky zadavatelské firmy zaměřené na vývoj softwarových produktů v oblasti technologie Voice over IP (VoIP) a poskytuje náměty k modifikacím stávajícího systému evidence softwarových chyb a jejich příčin.

ABSTRACT

This Bachelor`s thesis concerns with the process of fault detection in software programs. It contains hierarchically organized typology of software errors, gained by research of available literature and analysis of events responsible for influencing the software errors evolution gathered into a typology of factors inspired by cognitive psychology. It contains also comparison of software evolution techniques and methodologies from the point of product validation and verification.

The Bachelor`s thesis was written with respect to needs and requests of the commissioning company focusing on the Voice over IP (VoIP) software evolution. The thesis aims to suggest modifications of software errors database and error causes database.

KLÍČOVÁ SLOVA

Kvalita, software, testování, typologie, chyby, verifikace, metodiky, techniky

KEYWORDS

Quality, software, testing, typology, errors, faults, verification, methodology, technique

PROHLÁŠENÍ O ORIGINALITĚ

Prohlašuji, že jsem tuto bakalářskou práci zpracovala samostatně, dle rad a pokynů vedoucího práce, s využitím uvedené literatury.

V Brně dne 24. 5. 2013 Podpis: …….

PODĚKOVÁNÍ

Děkuji doc. ing. Branislavu Lackovi, CSc. za cenné rady a připomínky.

Děkuji Janu Sychrovi za náměty a poskytnutá data.

BIBLIOGRAFICKÁ CITACE

ČAPKOVÁ, H. *Kauzální analýza chyb řídících programů.* Brno: Vysoké učení technické v Brně, Fakulta strojního inženýrství, 2013. 62 s. Vedoucí bakalářské práce doc. Ing. Branislav Lacko, CSc..

OBSAH

SEZNAM POUŽITÝCH ZKRATEK

CBO	Coupling between Object Classes (spřažení mezi třídami)
CK metriky	Chidamber a Kemerer metriky
CMMI	Capability Maturity Model Integration (stupňovitý model zralosti)
DIT	Depth of Inheritance Tree (hloubka stromu dědičnosti)
DMADV	Define, Measure, Analyze, Design, Verify
DSDM	Dynamic Systems Development Model
FDD	Feature Driven Development (vývoj řízený užitnými vlastnostmi)
GUI	Graphical User Interface (grafické uživatelské rozhraní)
JAD	Joint Application Development
LD	Lean Development
LOC	Lines of Code (počet řádků ve třídě)
NOA	Number of Attributes (počet atributů)
NOC	Number of Children (počet potomků ve třídě)
NOM	Number of Methods (počet metod)
RAD	Rapid Application Development
RFC	Response for a Class (počet metod volaných třídou)
RUP	Rational Unified Process
SDLC	Software Development Life Cycle
SQuaRE	Software Product Quality Requirements and Evaluation
TDD	Test Driven Development (vývoj řízený testy)
TSP	Team Software Process
VoIP	Voice over IP
WMC	Weighted methods per class (váha metod pro třídu)
XP	Extrémní programování

SEZNAM TABULEK

SEZNAM GRAFŮ

SEZNAM OBRÁZKŮ

1 ÚVOD

Bakalářská práce se zabývá problematikou odhalování softwarových chyb. Jde o práci rešeršně-analytickou. Cílem je zpracovat typologii softwarových chyb (kap. 2), analyzovat faktory vedoucí ke vzniku softwarových chyb (kap. 3) a provést srovnávací studii různých postupů pro nalézání příčin chyb v programech (kap. 4) s ohledem na potřeby zadavatelské firmy. V této firmě jsou v současné době statistiky prováděny na intuitivně stanovených kategoriích a jedním z externích cílů této práce je navrhnout řešení, které by bylo použitelné pro kategorizaci softwarových chyb a jejich odhalování v této firmě. Příklady ilustrující kap. 1 a kap. 2 vycházejí ze stávajících statistik zadavatelské firmy, navržené typologie poskytují náměty k modifikacím stávajícího systému evidence chyb.

V úvodních kapitolách práce zaměřených na význam kvality softwaru se zabývám současným stavem z hlediska existujících norem a doporučení (kap. 2.1). V kap. 2.2 zpracovávám rešerši vedoucí k hierarchicky organizované typologii softwarových chyb. Dále se zaměřuji na přímé příčiny vzniku softwarových chyb (kap. 3.1) a nepřímé ukazatele rizika vzniku softwarových chyb (kap. 3.2). V části zaměřené na přímé příčiny využívám typologii příčin chyb inspirovanou kognitivní psychologií a aplikuji ji na jednotlivé vývojové fáze softwarového produktu. Tato typologie poskytuje tématu softwarových chyb širší kontext než úzce technické zaměření. V části týkající se nepřímých ukazatelů uvádím nejčastěji používané softwarové metriky a jejich význam pro prevenci vzniku softwarových chyb, zabývám se konceptem „čistého kódu“ a refaktorizací. V následující kapitole se zaměřuji na metodiky a techniky softwarového vývoje. Nejprve hodnotím výhody a nevýhody jednotlivých metodik softwarového vývoje z hlediska validace a verifikace (kap. 4.1 a 4.2), posléze zpracovávám návrh technik vhodných k odhalování příčin chyb zaznamenaných v kap. 3.1 (kap. 4.3 a 4.4). Tato část práce ústí v souhrnný návrh technik vhodných pro odhalování chyb během vývoje softwarového produktu, zpracovaný ve formě diagramu příčin a následků.

2 KVALITA SOFTWARU A TYPOLOGIE CHYB V ŘÍDÍCÍCH PROGRAMECH

2.1 Kvalita softwaru

Význam kvality softwaru v oblasti softwarového vývoje v posledních letech stále narůstá. Jakost softwaru není snadno kvantitativně měřitelná, proto se způsoby jejího měření stále stávají předmětem diskuze a jsou podrobovány zkoumání z hlediska jejich objektivity.

Kvalitu lze definovat jako soulad výsledného produktu se sjednanými požadavky a specifikacemi. Problém spočívá v tom, že v odpovídající fázi vývoje často není snadné definovat všechny komplexní požadavky, a není snadné předvídat komplikace vzniklé při nasazení programu do reálného prostředí. Jak systém narůstá a stává se složitějším, tyto obtíže se zvýrazňují a často je nutné dodatečně měnit specifikace požadavků, pozměňovat návrhy softwaru i testové sady. Při vývoji složitějších systémů prakticky není možné ve fázi specifikace požadavků kompletně předvídat všechny požadavky. Stejně tak ve fázi testování v terénu nejsou všechny chyby úplně popsány z hlediska okolností, které k nim vedly. [2]

Velká část chyb vzniká v důsledku nedostatečné, nekonzistentní nebo nepřesné specifikace. Další část chyb je důsledkem souhry neočekávaných vstupů v programu, se kterými vývojáři a testeři nepočítali. Proto je pro zajištění kvality software nutné klást důraz na následující vývojové fáze:

- Přesná specifikace požadavků
- Kvalitní provedení návrhu
- Provedení adekvátních testů [2]

Posuzování softwarové kvality se dotýká dvou složek: funkční a strukturální. Funkční softwarová kvalita reflektuje, do jaké míry software splňuje funkční požadavky a specifikace. Strukturální softwarová kvalita odkazuje k vlastnostem, které se funkčnosti přímo netýkají, ale jsou klíčové z hlediska distribuce a údržby softwaru (např. robustnost nebo administrace). Zatímco strukturální kvalitu lze hodnotit pomocí analýzy vnitřní struktury softwaru, zdrojového kódu a architektury, funkční kvalita je obvykle měřena testováním. [1]

2.1.1 Normy

Otázka kvality je pro firmy vyvíjející software zároveň otázkou jejich úspěchu či neúspěchu, proto je nasnadě klást si otázku, které faktory se na posuzování softwarové kvality podílejí. Měření softwarové kvality se opírá o řadu existujících norem. Z hlediska této práce lze považovat za klíčové následující normy:

- ISO/IEC 9000 Systémy managementu kvality (Quality management systems),
- ISO/IEC 9126 Softwarové inženýrství – Kvalita produktu(Software engineering — Product quality),
- ISO/IEC 25000 „SQuaRE“ Kvalita sofwarového produktu – Požadavky a hodnocení (Software Product Quality Requirements and Evaluation).

Pro komplexní přístup ke kvalitě softwaru by bylo pak nutno věnovat pozornost i řadě jiných norem, které však jsou zaměřeny mimo cíle mé bakalářské práce, např.

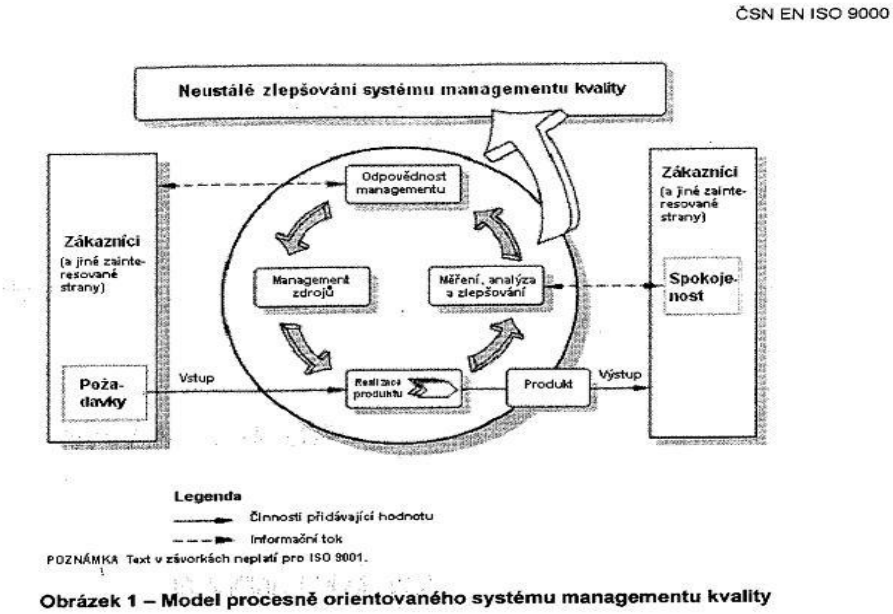
- ISO/IEC 12119 Softwarové inženýrství – Softwarové balíky – Požadavky na jakost a zkoušení (Software engineering – Software packages – Quality requirements and testing),
- ISO/IEC 14598 Softwarové inženýrství – Hodnocení softwarového produktu (Software engineering – Product quality evaluation),
- ISO/IEC 15504 Kvalita procesů vývoje, správy a nákupu software (Software Process Improvement and Capability determination),
- ISO/IEC 15939 Softwarové inženýrství – Proces měření softwaru (Software Engineering – Software Measurement Process).

Norma ISO/IEC 9000 [4]

Řada norem ISO 9000 stanovuje zásady managementu kvality. Zdůrazňuje, že „úspěšné vedení a fungování organizace vyžaduje, aby byla vedena a řízena systematickým a transparentním způsobem“. Cílem je neustále zlepšování výkonnosti organizace. Norma identifikuje osm zásad managementu kvality ke zvýšení výkonnosti:

a) Zaměření na zákazníka. Cílem je snaha rozumět současným i budoucím potřebám zákazníků, plnění jejich požadavků a snaha předvídat jejich očekávání.
b) Vedení a řízení lidí. Vedoucí mají vytvářet prostředí, ve kterém se lidé mohou plně zapojit při dosahování cílů organizace.
c) Zapojení lidí. Zdůrazňuje, že plné zapojení lidí umožňuje využít jejich schopnosti ve prospěch organizace.
d) Procesní přístup. Ve snaze dosáhnout požadovaného výsledku je účinné řídit činnosti a zdroje jako proces.
e) Systémový přístup k managementu. Organizace pracuje efektivněji, jsou-li procesy řešeny jako systém.
f) Neustálé zlepšování celkové výkonnosti organizace má být trvalým cílem.
g) Efektivní přístup k rozhodování se zakládá na faktech a informacích.
h) Vzájemně prospěšné vztahy mezi organizací a jejími dodavateli zvyšuje jejich schopnost vytvářet hodnotu.

Obr. 1 Model procesně orientovaného systému podle Normy ISO 9000 [4]



Obrázek 1 – Model procesně orientovaného systému managementu kvality

Norma ISO/IEC 9126 [5]

První normou, která specifikovala charakteristiky kvality softwaru, byla norma ISO 9126, které v 90. letech definovala šest kritérií softwarové kvality:

a) Funkčnost. Software má plnit definované požadavky.
b) Spolehlivost. Software je schopen udržovat určitou úroveň výkonu za stanovených podmínek po stanovenou dobu.
c) Použitelnost. Vyjadřuje, jak snadno se lze software naučit a používat.
d) Efektivnost. Software by měl efektivně využívat zdroje.
e) Udržovatelnost. Tato vlastnost se týká oprav defektů, rozšiřování funkcionalit atp.
f) Přenositelnost. Vyjadřuje, zda lze snadno přenést software na jiné hardwarové či operační systémy.

Tuto normu po roce 2005 postupně nahradila nová skupina norem řady ISO/IEC 25000, označovaná zkratkou SQuaRE.

SQuaRE [6]

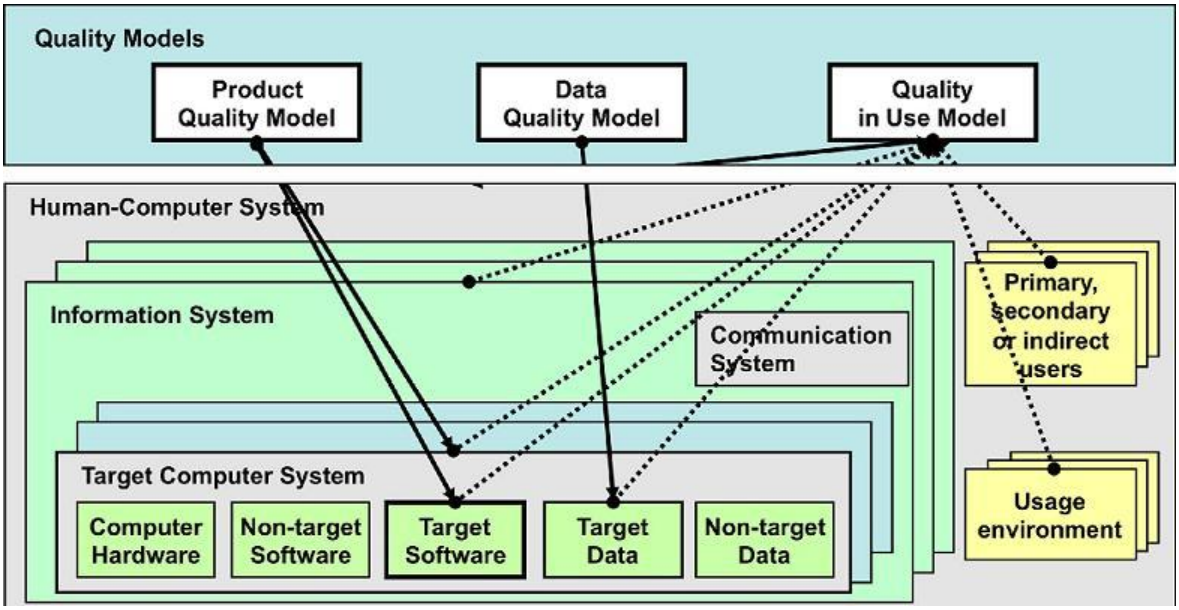
Řada norem 25000 stanovuje pět charakteristik týkajících se provozu softwaru, osm charakteristik ohledně vlastností softwaru a dva úhly pohledu na softwarová data. Tyto charakteristiky se dále člení na subcharakteristiky.

A) Kvalita provozu softwaru. Tato část popisuje vlastnosti ve vztahu k celému systému člověk – počítač a zahrnuje jak cílový softwarový produkt, tak cílový počítačový systém.
   a) Efektivnost
   b) Úspornost
   c) Spokojenost uživatele. Norma zmiňuje užitečnost, důvěru, pohodlí při používání.
   d) Zmírnění rizik, například ekonomických, bezpečnostních, environmentálních.
   e) Pokrytí kontextu. Vyjadřuje úplnost pokrytí nastálých situací a flexibilitu v používání.

B) Kvalita vlastností softwaru. Tento oddíl se zaměřuje na cílový softwarový produkt, týká se rovněž počítačového hardwaru, jiných softwarových produktů a dat. Vychází z normy ISO/IEC 9126, rozšiřuje ji a zpřesňuje.
   a) Funkční vhodnost. Software pokrývá požadavky úplně, správně a vhodným způsobem.
   b) Výkonová efektivnost. Popisuje časovou hospodárnost, míru využití zdrojů a kapacitu. Tuto charakteristiku norma ISO/IEC 9126 nezmiňovala.
   c) Kompatibilita. Týká se koexistence s jinými softwary a spolupráce s nimi.
   d) Použitelnost. Například snadnost učení se softwaru, odolnost vůči uživatelským chybám, estetičnost uživatelského interface, přístupnost.
   e) Spolehlivost. Týká se míry zralosti, dostupnosti, tolerance k chybám a znovuobnovení původního stavu.
   f) Bezpečnost. Vyjadřuje možnost utajení, celistvost systému, odpovědnost a potvrzení pravosti. Tato charakteristika je rovněž nová.
   g) Udržovatelnost. Software se sestává z modulů, je znovupoužitelný, umožňuje analýzu, lze jej modifikovat a testovat.
   h) Přenositelnost. Týká se přizpůsobivosti konkrétním požadavkům, snadnosti instalace, možnosti nahrazení jiným softwarem.

C) Kvalita softwarových dat. Data dělí následujícím způsobem:
   a) Nezávislá na systému
   b) Závislá na systému

Obr. 2 Cíle modelu kvality podle norem ISO 25000 [6]



2.1.2 Modely zajišťování softwarové kvality

Vedle ISO/IEC norem, které upravují problematiku softwarové kvality, se etablovala řada modelů zajišťování softwarové kvality, z nichž některé jsou široce přijímány jako vhodná doporučení při vývoji software, jiné jsou záležitostí konkrétních firem a jejich význam nepřekračuje hranice společnosti, která je vytvořila. Alespoň stručně zmíním následující modely:

- *Capability Maturity Model Integration (CMMI)*. Stupňovitý model zralosti vyvíjeného softwaru, definuje procesní oblasti realizace (řízení procesů, řízení projektů, návrh a realizace, podpůrné procesy), má 5 úrovní zralosti (počáteční, řízená, definovaná, kvantitativní, optimalizující). CMMI je podrobný model určený pro vývojové týmy, proto je v praxi snáze aplikovatelný než řada norem ISO 9000. Současná verze CMMI je blízká ISO 15504 (Software Process Improvement and Capability Determination, SPICE). Tento model je upřednostňován v USA, v Evropě převažují normy řady ISO. [31]
- *DMADV* je jedním z pilířů strategie Six Sigma. Na rozdíl od ostatních částí této strategie je metodika DMADV zaměřena na proces vývoje. Sestává se z částí definice – měření – analýza – návrh – ověření.
- *Seven Basic Tools of Quality*. Sedm základních nástrojů zvyšování kvality je pevně stanovený soubor především grafických technik, mezi které patří: diagram příčin a následků, kontrolní tabulka, histogram, Paretův diagram, korelační diagram, vývojový diagram a regulační diagram. Tento přístup je jednodušší než rozvinuté statistické metody, takže je možné jej použít i se základními znalostmi statistiky. Zdroje uvádějí, že využití metodiky je především ve výrobě, nicméně s určitými modifikacemi je jistě možné ji využít i při testování softwaru. [32]

2.1.3 Motivace softwarové kvality

Přelom tisíciletí přinesl do vývoje softwaru nové fenomény, které zvýraznily důležitost kvality softwarových produktů.

Jedním z nich je zvýšený tlak na flexibilitu softwarových firem: Snaha o co nejrychlejší reakce na požadavky zákazníků spolu se snahou o co nejnižší náklady vedly k potlačení tradičních, organizovaných metodik vývoje a k rozšíření agilních metod. Druhý vliv souvisí s rostoucí celosvětovou konkurencí. Ekonomická liberalizace, nárůst kapacit přenosových komunikačních kanálů a rostoucí počty vzdělaných pracovníků na celosvětových pracovních trzích vede ke zostření konkurence a dalšímu tlaku na kvalitu software. Třetím faktorem je zvýšený důraz na dodržování předpisů, který je reakcí na liknavost 90. let v této oblasti. [3]

Rozmach vzniku vzájemně neprovázaných předpisů a norem vytvořil roztříštěnou sadu doporučení, kterou se projekt SQuaRE snaží sjednotit. Při vytváření norem kvality softwaru čelí autoři řadě nesnází, které jsou spojeny nejen s vytvářením norem jako takových, ale také se samotným hodnocením kvality softwaru. Zmínit lze například následující [7]:

- Problematické formulování požadavků na kvalitu softwaru. Otázkou je, které charakteristiky lze považovat za podstatné pro hodnocení kvality softwaru, a které jsou spíše marginální.
- Problematický výběr vhodných vlastností (subcharakteristik) a jejich měření.
- Problematické vytvoření jednotné sady norem, které nejsou redundantní, jsou konzistentní, mají jednotnou terminologii, nepoužívají nejednoznačné pojmy (např. funkcionalita, kvalita v provozu).

Z výše uvedeného vyplývá, že vytvoření norem pokrývá oblast hodnocení kvality softwaru jen rámcově a dodržování norem je spíše dobrým odrazovým můstkem k vytvoření kvalitního softwaru než jeho zárukou.

Obecně je možno tvrdit, že zlepšení kvality softwaru lze v softwarových firmách dosáhnout:

- Systémovým přístupem ke kvalitě SW [24];
- Dobrým procesním řízením kvality softwaru s využitím norem ISO 9000 [4];
- Kvalitně zpracovaným postupem všech procesů testování softwaru [23].

V této situaci má smysl klást si otázku, jaké další kroky by měla firma vyvíjející software podniknout, aby výsledný software lépe obstál na trhu softwarových produktů. Odpověď se pokusím nalézt v následujících kapitolách.

2.2 Typologie chyb v řídících programech

Mluvíme-li o softwarových chybách, měli bychom rozlišovat mezi chybou ve smyslu nedostatku, závady, poruchy, zavinění (ang. fault) a chybou ve smyslu omylu (ang. error). Rozdíl v tomto pojetí navádí k jiným souvislostem; zatímco softwarové omyly vesměs zahrnují širší kontext vzniku softwarového produktu, softwarové nedostatky do značné míry evokují chyby programátorské a lze je tedy chápat jako podmnožinu obecnějších softwarových omylů. [14] Pro úplnost lze zmínit chyby ve smyslu výpadku, selhání (ang. failure) a chybu ve smyslu náhodného, neobjasněného selhání (ang. bug). Je zajímavé, že počet chyb v programu obvykle převažuje nad počtem pozorovatelných selhání: Ve výzkumu [17] na jedno selhání připadalo 2,87 softwarových chyb programu.

Rovněž způsob pojetí softwarových chyb a jejich příčin není jednoznačný, můžeme volit např. přístup

- Hierarchický (výčet chyb seřazených do logicky hierarchických kategorií), největší proporci budou mít chyby programátorské, protože na úrovni konkrétních chyb kódu lze rozlišit největší množství chyb, zatímco např. chyby specifikace lze obvykle shrnout do kategorie „chybná komunikace se zákazníkem“ a jsou do značné míry závislé na konkrétním softwarovém produktu,
- Chronologický (výčet chyb podle fáze vývoje, ve které se objevují),
- Abecední (výčet chyb seřazených podle abecedy),
- Frekvenční (řazení podle četnosti chyb, eventuelně podle četnosti vzhledem k fázím vývoje produktu), z tohoto pohledu je často nejvýznamnějším viníkem vzniku chyb nevhodná specifikace [8],
- Kauzální (chyby podle předpokládaných příčin),
- Manifestační (chyby podle projevů), do značné míry závisí na konkrétním softwarovém produktu.

V této kapitole volím přístup **hierarchický** a zaměření na chyby ve smyslu omylů, tedy užší pojetí softwarových chyb. Mým cílem bylo shromáždit co největší množství různorodých typů chyb, jak jsou uváděny v literatuře, a roztřídit je do sourodějších kategorií.

V kapitole 3 „Příčiny vzniku chyb v řídících programech“ k chybám přistupuji **chronologicky** podle fáze vývoje a snažím se je uchopit ve smyslu nedostatků obecně, tedy v širším pojetí. V této podkapitole jsem použila typologii příčin chyb autorů Walia – Carver [14], kteří ji aplikovali na chyby vzniklé ve fázi specifikace požadavků, a pokusila jsem se tuto typologii aplikovat i na další fáze projektu. Tento přístup jsem zvolila proto, že příčiny chyb jsou v každé fázi vývoje zásadně odlišné a vytvořením obecného teoretického rámce pro jejich kategorizaci lze dospět k odhalení příčin, které na první pohled nejsou zjevné. Kauzální přístup postupuje naopak, k zaznamenaným chybám jsou dohledávány předpokládané konkrétní příčiny.

Při zpracování typologie chyb byla použita zejména literatura [9][10][11][12][13].

A. Chyby organizační
- *Chyby v zadání a specifikaci* obvykle znamenají, že specifikace neodpovídají očekáváním zákazníků a potřebám uživatelů, jsou nejednoznačné, neúplné nebo nekonzistentní.
- *Chyby implementace*, např. chybná analýza zadání nebo chybně naplánovaný postup.
- *Chyby dokumentace* znamenají, že funkcionalita programu není v dokumentaci dostatečně popsána, manuály jsou matoucí nebo chybí, není k dispozici kvalitní nápověda.
- *Chyby organizace verzí* programu mohou např. znamenat zakomponování starých verzí komponent do programu.
- *Chyby třetí strany* označují chyby v softwaru, pro nějž nemáme přístup ke zdrojovému kódu.

B. Chyby technické

a) *Chyby testování*. Např. nesprávně navržené testy, neúplné testy, špatně prováděné testy.

b) *Chyby v programátorském kódu*

i. *Chyby nefunkční (behaviorální)* obvykle nelze diagnostikovat přímo, ale častěji podle nepřímých veličin; např. ukazatelem spolehlivosti může být chybovost (nepřímá úměra).
- *Zátěžové chyby* značí, že program selhává při velké zátěži (zpracování velkého množství dat, běh po dlouhou dobu).
- *Chyby výkonu* značí, že běh softwaru je příliš náročný pro daný hardware.
- *Chyby souběhu*, ke kterým dochází při paralelním běhu s jinými programy, patří k velmi obtížně odhalitelným i testovatelným.
- *Chyby rychlosti odezvy* jsou spíše problémem než chybou v užším slova smyslu.
- *Chyby spolehlivosti* obvykle znamenají, že program se nechová konzistentně podle očekávání a nereaguje vždy stejným způsobem.
- *Chyby bezpečnosti* mohou znamenat, že k jistým částem programu mají přístup neoprávněné osoby, nebo k němu nemají přístup osoby oprávněné.
- *Problémy testovatelnosti* odkazují k cyklomatické složitosti kódu, která měří komplexnost rozhodovací logiky a lze ji určit jako počet podmínek v rámci jedné funkce. Vyšší cyklomatická složitost obvykle znamená, že kód se hůře testuje, udržuje, a je náchylnější k chybám.

ii. *Chyby funkční*
- *Chyby v uživatelském prostředí* mohou znamenat například, že existuje dlouhá časová odezva, nevhodné rozvržení GUI, nekonzistence uživatelského rozhraní, nadbytečnost prvků, přílišná složitost v používání, učení nebo navigaci, grafická nepřehlednost, neefektivnost (program uživateli vnucuje neefektivní způsoby řešení, které nevedou k cíli), rigidita (program vede k uvažování pevným nebo nepřirozeným způsobem).
- *Chyby funkcionality* mohou znamenat, že program nedělá to, co by měl dělat, nebo to dělá zmatečně či neúplně, nebo dělá něco, co by dělat neměl. Představa, co by program měl nebo neměl dělat, závisí na očekávání uživatele a měla by být pokryta ve specifikacích.

Následující výčet se týká funkčních chyb v programátorském kódu. Nejedná se o vyčerpávající seznam, spíše o shrnutí nejčastěji uváděných typů chyb v použité literatuře. Zároveň se jednotlivé typy chyb nevylučují, např. překlep může způsobit chybu v logickém výrazu a tím vést k zacyklení a selhání programu.

- *Opomenutí* vzniká v důsledku chybějícího kódu potřebného pro celkovou funkcionalitu, např. pro volání funkce, kontrolu průběhu programu, chybějící výpočetní či logický výraz.
- *Překlepy* jsou nejčastější a obvykle nejsnáze odhalitelnou chybou v programovacím kódu, často zachyceny již překladačem. Např. dvě rovnítka namísto jednoho, nadbytečné či chybějící středníky, špatně uzavřené závorky.
- *Procesní chyby* se týkají situací, s nimiž programátor nepočítal: Existují např. chyby speciální, kdy při běhu programu dochází k chybám za určitých specifických okolností, např. při prvním běhu programu nebo při resetování. Dále může docházet k chybám běhu, kdy v jistém kroku program provede nesprávnou operaci, zacyklí se, uvázne nebo selže.
- *Chyby kontroly toku* může způsobit např. přidání nového zdrojového bloku kódu, smazání potřebného bloku kódu (např. volané funkce), vložení chybné podmínky pro provádění bloku kódu, nesprávné pořadí provádění příkazů, přidání procedury nebo funkce, odstranění procedury nebo funkce, volání externí procedury nebo funkce s nesprávným argumentem, chybná návratová hodnota, chybný logický operátor, chybný ukazatel.
- *Chyby v programovacím jazyce*. Např. nulový ukazatel, ukazatel na neplatnou adresu.
- *Chyby zpracování výjimek* se vztahují k nepředpokládaným situacím, které mohou v programu nastat, a s nimiž se nepočítalo, může se jednat např. o zpracování neplatného uživatelského vstupu.
- *Chyby hraničních podmínek* nastávají např. při překročení hraničních hodnot nebo přetečení zásobníku.
- *Výpočetní a logické chyby* se týkají výpočetních a logických výrazů a mohou znamenat např. zaokrouhlovací chyby, znovupoužití registrů, nesprávnost rovnic, chybnost algoritmů.
- *Chyby funkcí* jsou důsledkem chybné operace s funkcemi, např. volání funkce s chybným parametrem nebo volání jiné funkce, než je potřeba.
- *Datové chyby* mohou být jednoduché, jako např. definování proměnné s nevhodným datovým typem, nebo složitější, jako např. chyby inicializace určité funkce.
- *Chyby v proměnných, vlastnostech, metodách, třídách*, jako např. chybné deklarace nových či předefinování existujících proměnných, smazání proměnných, změna hodnot v existujících proměnných, použití metody s nesprávným návratovým typem nebo nevhodné použití překrytelné metody.
- *Chyby v konstantách*, např. chybná definice či redefinice nebo smazání.
- *Chyby přístupu ke zdrojům* mohou znamenat, že alokovaná paměť není inicializována nebo uvolněna, kvůli chybnému časování dochází k deadlockům, dochází k přetečení zásobníku.
- *Chyby rozhraní* mohou vznikat např. v důsledku nedostatečného transportu dat nebo nepotřebných návratových hodnot
- *Falešné chyby* vznikají přehlcením kódu a jejich řešením je odstranění přebytečných vyjádření.
- *Chyby statického kódu* vznikají např. v důsledku neproběhnutých změn v kódu po kompilaci nebo prvním provedení programu.
- *Hardwarové chyby* se vztahují k nesprávnému ovládání hardwaru softwarem, program např. posílá data na nesprávná zařízení.

Podle předchozího výčtu by se mohlo zdát, že největší procentuální zastoupení v celkovém počtu chyb mají chyby v programovém kódu. Obvykle tomu tak není. Chyby programového kódu lze ve srovnání s ostatními chybami snáze klasifikovat, protože se týkají pravidel programovacího jazyka, zatímco ostatní typy chyb závisejí do značné míry na konkrétním softwarovém projektu.

Ron Patton uvádí, že „příčinou většiny [chyb] nejsou omyly při programování. Byla provedena řada studií nad projekty různé velikosti – od velmi malých po extrémně velké – a jejich výsledky jsou vždy stejné. U softwarových chyb je příčinou číslo 1 specifikace.“ [8, s. 15]

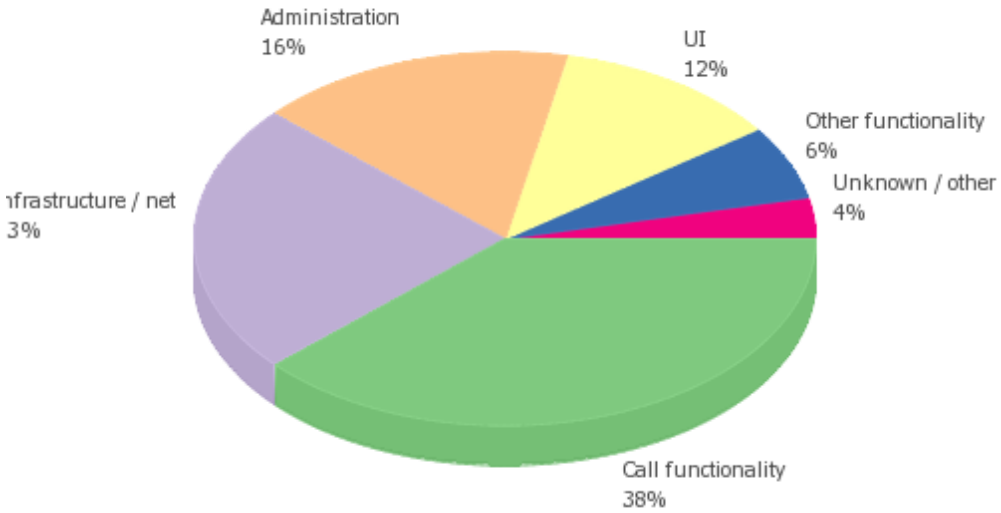
Problém s chybami vzniklými ve fázi specifikace narůstá, pokud vezmeme v potaz, že s postupem času rostou několikanásobně náklady na opravu chyb. Podle Pattona [8, s. 17] roste výše nákladů logaritmickou řadou; chyba odhalená ve fázi specifikace bude stát desetkrát více při odhalení během kódování, stonásobně při odhalení během testování, a její cena bude až tisíckrát vyšší, najde-li ji zákazník v hotovém produktu.

Ukazuje se, že chyby nejsou ve zdrojovém kódu rozděleny rovnoměrně. Většina chyb se soustředí v několika vysoce chybových třídách a rutinách. McConnell uvádí, že 80% chyb najdeme ve 20% tříd nebo rutin a 50% chyb se soustředí v 5% tříd. Tyto vysoce chybové třídy a rutiny jsou navíc těžko odhalitelné a těžko opravitelné: 20% chyb vyžaduje 80% prostředků na opravu všech chyb. Vysokou chybovostí se obvykle vyznačují třídy obsahující příliš složité rutiny, proto by měla být aktivita soustředěna na rozpoznávání, přepracovávání a přepisování rutin s vysokou složitostí a chybovostí. [16, s. 526] (metody rozpoznávání složitosti viz kap. 3.2)

Podle jednoho výzkumu [17], zohledňujícího kromě statických metrik také časovou osu, se ukázalo, že problematické jsou třídy, které se v systému vyskytují po dlouhou dobu, a počet chyb v nich narůstá rychlejším tempem, než je obvyklé v daném systému.

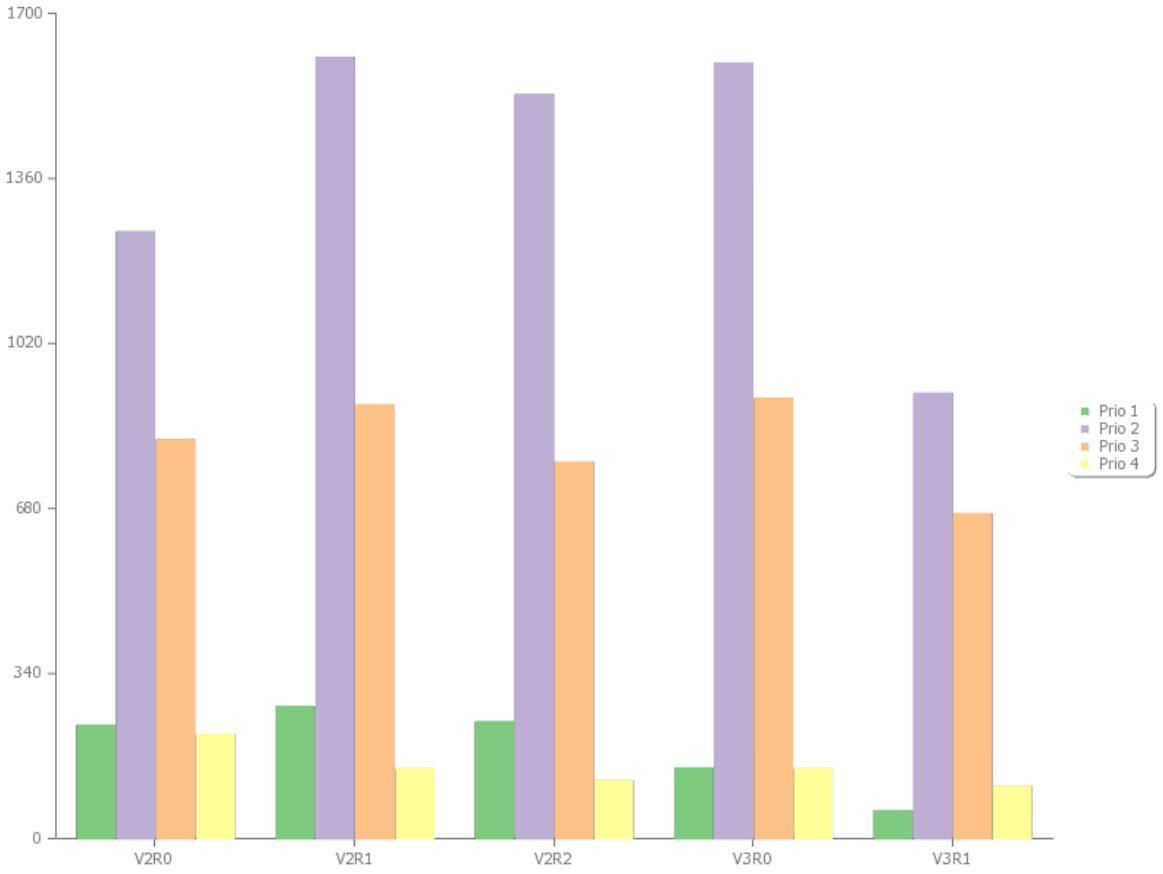
Přiložený graf ilustruje statistické rozložení chyb v oblastech kódu jednoho z projektů zadavatelské firmy. Jedná se o chyby problematické, které mají tendenci se znovuobjevovat navzdory pokusům o opravy, nebo se objevují jako vedlejší efekt jiných oprav.

Graf 1 Problematické chyby podle oblastí kódu



Ve středně velkých a velkých firmách jsou dokumentované chyby obvykle rozdělovány podle závažnosti do několika tříd (nejčastěji do čtyř). Rozdělení chyb podle závažnosti v různých verzích projektu zadavatelské firmy ilustruje následující graf.

Graf 2 Reportované chyby podle závažnosti na různých verzích projektu. S časovým postupem projektu lze sledovat úbytek chyb priority 1 a posléze i celkový úbytek reportovaných chyb.



3 PŘÍČINY VZNIKU CHYB V ŘÍDÍCÍCH PROGRAMECH

3.1 Přímé příčiny vzniku chyb

Příčiny vzniku chyb softwarového produktu zčásti naznačila již předchozí kapitola. Důležitým zjištěním je skutečnost, že prakticky nelze vytvořit klasifikaci vzájemně se vylučujících typů chyb, protože chyby jsou často provázány na různých úrovních, kdy chyby jednoduché, např. opomenutí, se v důsledku mohou projevovat jako chyby složité, např. chyby souběhu s jinými programy.

Při hledání prvotních příčin chyb se věc ještě více komplikuje. Např. opomenutí jisté funkce v programu může být způsobeno tím, že nebyla zmíněna ve specifikaci, nebo tím, že ve specifikaci sice zmíněna byla, ale programátor nebyl s touto specifikací seznámen, což může způsobeno špatnou organizací, časovou tísní, nebo neznalostí programátora. Při hledání důvodu, proč nebyla funkce zmíněna ve specifikaci, můžeme nalézt opět více různých příčin.

Otázka „jak se chyba projevuje“ je teď propojena s otázkou „z jakého důvodu chyba vznikla“. Odpovědi na tyto otázky jsou důležité z hlediska včasného zlepšování kvality produktu. Ukazuje se, že podobné typy chyb se projevují obdobně a vyžadují podobné úsilí o nápravu, proto je přínosné sdružovat chyby podle jejich vlastností a vytvářet taxonomie. Avšak přestože jsou kategorie chyb užitečné, nelze podle nich nalézt veškeré chyby. Pomocí taxonomií chyb lze jen obtížně identifikovat nedorozumění vzniklé na jiných úrovních vývoje softwarového produktu. [14]

Hlavním nedostatkem přístupů založených na kategorizaci chyb je podle autorů Walia – Carver [14] absence přímého vztahu mezi zdrojem problémů (error) a jeho projevem (fault). Nepřínosné je krátkodobé řešení zaměřené na potlačení symptomů chyby, aniž byla nalezena primární příčina. Řešení se autoři pokoušení nalézt ve spojení kognitivní psychologie se znalostmi o softwarové kvalitě, a to ve vztahu k chybám vzniklým ve fázi specifikace požadavků.

Tab. 1 Popis tříd nedostatků týkajících se požadavků [14]:

Typ chyb	Třída chyb	Popis
Lidské selhání	Komunikace	Chybějící nebo špatná komunikace mezi zainteresovanými osobami
	Účast	Chybí důležití účastníci jednání
	Znalost problematiky	Autoři požadavků nemají znalosti nebo zkušenosti s problematikou
	Specifická znalost aplikace	Autoři požadavků neznají specifické aspekty aplikace
	Průběh procesu	Požadavky jsou nedostatečně vysvětleny vývojářům
	Jiné znalosti	Jiné chyby vzniklé z důvodu omezených znalostí autorů požadavků
Chyby procesu	Nevhodná metoda dosahování cílů	Výběr nevhodných nebo nesprávných metod, technik a přístupů k dosažení cílů
	Řízení	Nedostatky v procesu řízení projektu
	Předávání znalostí	Nedostatky v procesu objasňování požadavků
	Analýza	Nedostatky při analýze požadavků
	Sledovatelnost	Neúplná sledovatelnost požadavků
Dokumentační chyby	Organizace	Problémy s organizací požadavků při dokumentaci
	Absence standardů	Problémy z důvodu chybějících dokumentačních standardů
	Specifikace	Obecné chyby v dokumentaci nezávislé na znalostech autorů požadavků

Obecná klasifikace těchto autorů je přínosná i ve vztahu k chybám obecně. Pro účely této kapitoly můžeme vývoj produktu rozdělit dle Vodopádového modelu do následujících fází: [13]

1. Specifikace požadavků
2. Návrh a implementace
3. Programování
4. Testování
5. Údržba

Na jednotlivé fáze jsem aplikovala třídy chyb, které používají Walia – Carver ve své taxonomii. Výsledkem je konkretizovaná typologie zohledňující příčiny chyb v jednotlivých fázích vývoje. Její výhodou je, že z globálního pohledu upozorňuje na možná problematická místa vývoje produktu. Nevýhodou je, že tato typologie zůstává dosti obecná a nepostihuje konkrétní příčiny kritických chyb.

V řádcích jsou třídy chyb navržené autory Walia – Carver, ve sloupcích fáze projektu, v polích tabulek konkretizované příčiny chyb. Pole tabulky obsahují příklady, nikoliv vyčerpávající výčet projevů chyb způsobených danou příčinou v dané fázi projektu.

Tab. 2 Příčiny chyb ve fázi specifikace požadavků [14][15]

Třída chyb	Specifikace požadavků
Komunikace	problematická komunikace zákazníka a developerské firmy
Účast	chybí klíčoví uživatelé
Znalost problematiky	autoři požadavků neznají dobře problematiku
Specifická znalost aplikace	autoři požadavků neznají specifika aplikačního programového rozhraní
Průběh procesu	požadavky nejsou jasně vysvětleny sw architektovi nebo vývojářům
Jiné znalosti	autoři požadavků nerozumí obchodním požadavkům zákazníka
Nevhodná metoda dosahování cílů	vágní požadavky v situaci vyžadující přesné definice
Řízení	časový tlak
Předávání znalostí	požadavky nejsou systematicky vysvětleny vývojářům
Analýza	analýza požadavků není správná nebo úplná
Sledovatelnost	body požadavků nejsou jasně sledovatelné v rámci vývoje
Organizace	dokumentace požadavků ve firmě je chaotická
Absence standardů	nejsou k dispozici standardy ohledně definování požadavků
Specifikace	požadavky jsou chybně zdokumentovány

Tab. 3 Příčiny chyb ve fázi návrhu a implementace [16][3][15]

Třída chyb	Návrh a implementace
Komunikace	špatná komunikace autorů požadavků a sw architekta
Účast	v případě týmu architektů chybí klíčová osoba
Znalost problematiky	návrh je příliš složitý nebo naopak zjednodušující
Specifická znalost aplikace	volba nevhodného programovacího jazyka
Průběh procesu	návrh probíhá chaoticky namísto pravidel
Jiné znalosti	volba nevhodných tříd nebo objektů
Nevhodná metoda dosahování cílů	snaha o všezahrnující návrh během návrhové fáze projektu
Řízení	nezahrnutí neplánované práce do časového plánu
Předávání znalostí	v návrhu nejsou záhrnuty všechny zákaznické scénáře
Analýza	při analýze objektů je pominuto např. kritérium bezpečnosti
Sledovatelnost	prvky návrhu nejsou sledovatelné ve zdrojovém kódu
Organizace	dokumentace nezohledňuje fázi nasazení systému
Absence standardů	pro vytvoření návrhu neexistují standardní postupy
Specifikace	návrh není zdokumentovaný, nebo naopak dokumentace návrhu pohlcuje zbytečně mnoho úsilí

Tab. 4 Příčiny chyb ve fázi programování [16]

Třída chyb	Programování (v tomto sloupci dokument = program)
Komunikace	špatná komunikace mezi sw architektem, projektovým manažerem, vedoucím týmu, programátory
Účast	absence programátorů (manažerů, vedoucích) s klíčovými znalostmi
Znalost problematiky	nedostatečné zkušenosti manažerů (vedoucích) s podobnými projekty
Specifická znalost aplikace	nedostatečné zkušenosti programátorů s daným programovacím jazykem
Průběh procesu	nejasně rozdělené povinnosti v rámci programátorského týmu
Jiné znalosti	překlepy, opomenutí v programovém kódu
Nevhodná metoda dosahování cílů	nesprávný typ nebo řízení programového cyklu, chybné aserce
Řízení	časový stres, neplánovaná geografická roztříštěnost programátorského týmu
Předávání znalostí	opomíjení výhod programování ve spolupráci (párové programování, revize kódu)
Analýza	nevhodné stanovení výjimek v programu, nevhodné datové typy
Sledovatelnost	chaotická organizace komponent a verzí programu
Organizace	opakující se činnost namísto skriptů (maker, dávkových souborů)
Absence standardů	nejednotné pojmenovávání proměnných a konstant
Specifikace	nedostatečné ladění a refaktorizace

Tab. 5 Příčiny chyb ve fázi testování [8]

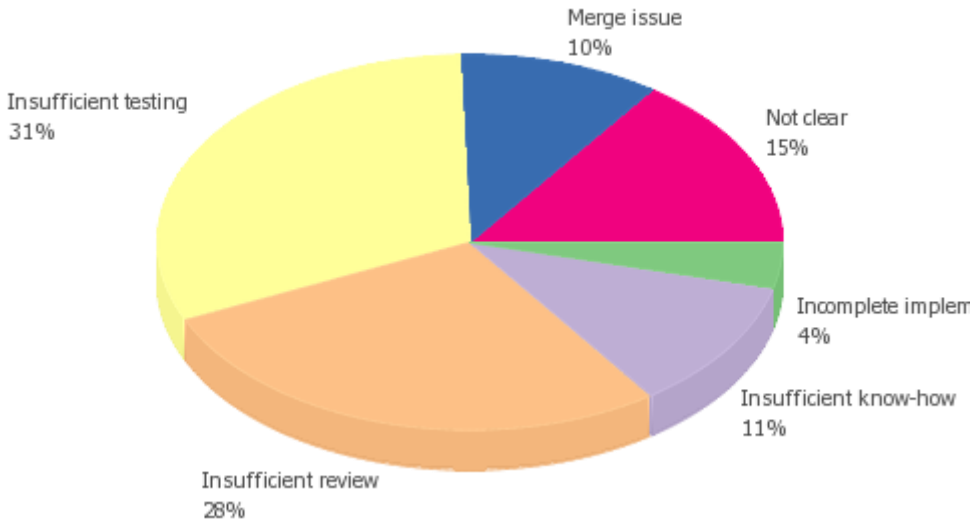
Třída chyb	Testování (v tomto sloupci dokument = testovací scénář nebo testovací sada)
Komunikace	špatná komunikace mezi testery a ostatními členy týmu
Účast	absence dostatečného počtu testerů
Znalost problematiky	testy nejsou správně navrženy s ohledem na danou problematiku
Specifická znalost aplikace	testeři nemají dost zkušeností s daným sw produktem
Průběh procesu	nejsou zahrnuty důležité způsoby testování (regresní, zátěžové, výkonové, kompatibility)
Jiné znalosti	testeři neprovádějí testy odpovídajícím způsobem
Nevhodná metoda dosahování cílů	nevhodný postup testování (např. pouze white-box testování, black-box opomíjeno)
Řízení	nedostatečné vybavení pro testování
Předávání znalostí	testeři neznají obvyklé zákaznické scénáře a konfigurace
Analýza	při analýze testovacích scénářů je zvolena nevhodná sada testovacích případů (nepokryjí podstatné vlastnosti produktu)
Sledovatelnost	nelze dohledat, na které verzi produktu byly testy prováděny
Organizace	nalezené chyby nejsou řádně zdokumentovány
Absence standardů	provedené testy nejsou dokumentovány jednotným způsobem
Specifikace	testovací sady nejsou jasně propojeny s požadovanými vlastnostmi produktu

Tab. 6 Příčiny chyb ve fázi údržby [13]

Třída chyb	Údržba
Komunikace	problematická komunikace mezi zástupci firmy a zákazníkem
Účast	nedostatek pracovníků zákaznické podpory
Znalost problematiky	nedostatečné znalosti pracovníků zákaznické podpory
Specifická znalost aplikace	nedostatek zkušeností pracovníků zákaznické podpory s produktem
Průběh procesu	nedostatečně flexibilní reakce na zákaznické stížnosti (pomalé řešení ohlášených problémů)
Jiné znalosti	zákaznická podpora nesprávně vyhodnotí závažnost ohlášeného problému
Nevhodná metoda dosahování cílů	chaotické doprogramovávání a opravování verzí programu
Řízení	neadekvátní frekvence vydávání opravných patchů a verzí
Předávání znalostí	zákazníci nejsou informováni o specifikách instalace a údržby programu
Analýza	analýza zákaznických stížností nemá přednost před vyvíjením nových produktů
Sledovatelnost	nelze dohledat zdrojový kód k zákaznickým verzím programu (z hlediska vývojářské firmy může jít o "starší" verze)
Organizace	uživatelské manuály nejsou zavčas k dispozici
Absence standardů	uživatelský manuál je nejednotný (např. formálně nebo terminologicky)
Specifikace	uživatelský manuál obsahuje chyby

Přiložený graf ilustruje příčiny problematických chyb (opakovaných nebo vzniklých opravami jiných chyb) v jednom z projektů zadavatelské firmy.

Graf 3 Problematické chyby podle příčin.



3.2 Nepřímé ukazatele rizika vzniku chyb

Až dosud jsem se zabývala softwarovými chybami a přímými příčinami jejich vzniku. Při zaznamenávání faktorů, které mohou vést ke vzniku chyb v programu, je účelné zabývat se rovněž nepřímými ukazateli rizika vzniku chyb. Pro tyto účely jsou ve vývoji obvykle využívány softwarové metriky.

Je důležité poznamenat, že vzájemný vztah mezi softwarovými metrikami a chybovostí softwaru nemusí být a obvykle také není kauzální a v mnoha případech jde pouze o statistickou shodu [17]. Mezi metrikami a výskytem chyb navíc zdaleka není přímá úměra. Přesto jsou metriky důležitým ukazatelem z hlediska predikce vzniku chyb. Mohou se např. kombinovat, spojovat nebo z nich lze počítat vážený průměr.

Příklady metrik zdrojového kódu [13., s. 156-164, 264][16, s. 472]:

- *Cyklomatická složitost* (cyclomatic complexity) – určení lineárně nezávislých cest v rámci funkce. Autorem je Thomas Cabe. Hodnota 0-5 na rutinu je pravděpodobně v pořádku, 6-10 – rutina by měla být zjednodušena, 11 a více – část rutiny by měla být přesunuta do samostatné rutiny. Hodnota cyklomatické složitosti se stanoví např. tímto způsobem: Začneme číslem 1 za přímý průchod rutinou – přidáme 1 za každé z klíčových slov *if, while, repeat, for, and, or* nebo jejich ekvivalenty – přidáme 1 za každou alternativu v přepínači.
- *Halsteadovy metriky* měří výskyt syntaktických prvků v programu. Zjišťuje se počet jedinečných operátorů, počet jedinečných operandů, celkový počet výskytů operátorů a celkový počet výskytů operandů. Jejich kombinací získáme např. metriku délky programu nebo programové náročnosti.
- *Metriky živosti kódu* popisují počet změn, ke kterým dojde v souboru či modulu během určitého časového období. Kromě vývoje nových funkcionalit se na nich podílejí rovněž opravy nalezených chyb. Vysoké procento oprav ve skutečnosti chybu neodstraní, ale může navíc vytvořit jinou. Je pak nutné provést další změny (a tím opět zvýšit živost kódu). Je-li kód složitý, tento postup se opakuje i několikrát, než jsou veškeré chyby odstraněny. Celkovou živost kódu lze spočítat např. součtem následujících hodnot:
  - Počet změn – udává, kolikrát byl soubor změněn
  - Počet přidaných řádků – udává, kolik řádků bylo přidáno od zvoleného momentu
  - Počet smazaných řádků – udává, kolik řádků bylo smazáno od zvoleného momentu
  - Počet změněných řádků – udává, kolik řádků se během sledovaného období změnilo
- *Objektově orientované metriky* se vztahují k třídám a jejich struktuře v jazycích jako C++, Java a C#. Nejznámější z nich, vytvořené autory Chidamber a Kemerer (CK metriky) zohledňují např.
  - Velikost třídy (LOC) – počet řádků ve třídě
  - Váhu metod pro třídu (WMC) – celková složitost metod ve třídě
  - Počet metod volaných třídou (RFC)
  - Hloubku stromu dědičnosti (DIT) – z kolika tříd tato třída dědí
  - Počet potomků ve třídě (NOC)
  - Spřažení mezi třídami (CBO) – kolikrát tato třída používá metody nebo atributy jiné třídy
  - Počet atributů (NOA)
  - Počet metod (NOM)

Za povšimnutí stojí, že podle výzkumu [21] se pozitivní korelace mezi CK metrikami a počtem chyb projevila pouze pro LOC, parametry WMC a CBO vliv neměly.

Podle jiného výzkumu [17] vliv nemají ani parametry založené na dědičnosti. Podle tohoto výzkumu mají pro predikci chyb v systému největší váhu parametry RFC, LOC a WMC.

Vedle metrik zdrojového kódu lze vytvářet také metriky založené na chybách. Tyto obvykle vycházejí z databáze pro sledování chyb a jsou užitečné pro vyhodnocování kvality softwaru. Tyto metriky mohou svádět k tomu, aby byly použity jako měřítko produktivity programátorů a testerů, nebo k nastavování latěk počtu chyb přiřazených jednomu programátorovi. Samotné statistiky chyb by ale neměly být používány k měření výkonu jednotlivce, protože jsou ale ovlivněny mnoha dalšími faktory (např. složitostí testované funkcionality, schopnostmi autorů kódu, kompletností specifikací).

Příklady metrik chybovosti [13., s. 203-204]:

- Procento opravených chyb – poměr počtu opravených chyb k neopraveným.
- Počet chyb na jazyk – umožňuje zjistit náklady na testování lokalizovaných verzí.
- Rychlost odhalování chyb v průběhu času – příliš vysoké nebo příliš nízké hodnoty mohou indikovat problem a měly by být objasněny.
- Počet chyb podle oblastí kódu nebo funkcí– setříděný seznam funkcí s nejvíce nalezenými chybami naznačuje, kudy zaměřit testování.
- Chyby podle závažnosti – chyby kategorizované jako nejzávažnější by měly být nacházeny v úvodních fázích vývoje.
- Podle místa výskytu – porozumění tomu, ve kterých částech softwaru jsou nacházeny chyby, může odhalit rizikové oblasti produktu.
- Míra reaktivace chyb – naznačuje, jak kvalitní opravy programátoři vytvářejí. Ke konci projektu se zpravidla zvyšuje kvůli rychlejšímu opravování chyb.
- Počet chyb podle typu testovací aktivity – zjištění, které testovací postupy vedou k nalezení velkého množství chyb, může zaměřit testování tímto směrem. Jedná se např. o integrační testy, automatizované testy, regresní testy, akceptační testy apod.
- Průměrný čas pro vyřešení – měří rychlost odezvy týmu.
- Průměrný čas pro uzavření – měří celkovou reakci týmu na odhalené chyby a čas potřebný k provedení celého procesu zpracování chyb.

Během vývoje softwarových produktů se každá vývojářská firma snaží o minimalizaci softwarových chyb takovým způsobem, aby výsledný produkt vykazoval co nejnižší míru chybovosti. Důležitou roli v tomto procesu hrají prevence chyb a odhalování již vzniklých chyb. Součástí prevence je snaha vytvářet kód, který je méně náchylný k chybám.

Vedle jasně pojmenovatelných kvalitativních příčin a statisticky uchopitelných kvantitativních příčin chybovosti kódu můžeme zmínit další koncept, který není tak snadno definovatelný, zato má praktické využití v práci programátorů. Martin [18] mluví o *čistém kódu*, McConnell [16] o *dokonalém kódu*. Martin uvádí několik definic čistého kódu od několika autorů, které podtrhují podstatu věci. Čistý kód je "elegantní a účinný… jednoduchý a přímočarý… čitelný a srozumitelný… pečlivě napsaný… projde všemi testy, neobsahuje žádná opakování kódu, vyjadřuje veškeré myšlenky návrhu, minimalizuje počet entit" [18, s. 30-33]. V dalším textu se zaměřuje na čtyři posledně jmenované vlastnosti a zabývá se úvahami, které procesy mohou podpořit vznik takto popsaného kódu. Dochází k závěru, že se jedná o dva zásadní principy:

1. Vytváření testovatelného systému (s malými a jednoúčelovými třídami) a co nejobsažnější testování.

2. Refaktorizace kódu s ohledem na návrh systému. Měly by být odstraněny shodné části kódu, podobné části přepracovány k ještě větší podobnosti a refaktorizovány. Kód by měl jasně vyjadřovat záměr – volit srozumitelné názvy, vytvářet malé třídy a funkce. Popis testů by měl být rovněž srozumitelný. Snaha o malé třídy a funkce by zároveň měla být provázena snahou o malý počet entit, protože jde však o do značné míry protichůdné procesy se snahou o hledání kompromisu, stojí tato premise na posledním místě jmenovaných vlastností.

Aby nebyla refaktorizace kontraproduktivní (nezpůsobovala vznik dalších chyb), mělo by jít o cílevědomou změnu aplikovanou po malých dávkách. Takto může být klíčovou strategií podporující stabilní zlepšování kvality program během údržby a může zabránit postupnému odumírání softwaru. [16, s. 578]

McConnell uvádí několik zásad bezpečné refaktorizace [16, s. 586-587]:

- Zálohování by mělo být provedeno před začátkem refaktorizace a také během kontrolních bodů refaktorizace.
- Refaktorizace by měla probíhat po malých krocích
- Vytvoření seznamu kroků, které mají být vykonány. Pokud se uprostřed refaktorizace zjistí, že v jejím rámci je třeba refaktorizovat další část kódu, měl by být vytvořen další seznam a tato druhá refaktorizace by měla počkat na dokončení první, je-li to možné.
- Nastavení citlivosti překladače během refaktorizace na nejvyšší úroveň.
- Regresní testování změněného kódu.
- Bezprostřední kontrola provedených změn. Při první úpravě kódu je 50% pravděpodobnost vytvoření chyby. Nevyšší pravděpodobnost vytvoření chyby je tehdy, pracuje-li programátor zároveň s 5-10 řádky kódu. Pracuje-li s jedním řádkem nebo naopak s větší částí kódu, pravděpodobnost vytvoření chyby klesá. Doporučení: zacházet s jednoduchými refaktorizačními změnami tak, jako by bylo složité, a používat při nich kontroly jinou osobou.
- Zvýšená opatrnost při rizikových refaktorizacích se závažným dopadem.
- Pokud restrukturalizujeme velkou část kódu, je třeba zvážit, zda by nebylo vhodné změnit návrh a upravovanou část pak znovu implementovat.

4 SROVNÁNÍ POSTUPŮ ODHALOVÁNÍ PŘÍČIN CHYB ŘÍDÍCÍCH PROGRAMŮ

Při odhalování příčin vzniku chyb můžeme problém uchopit z různých hledisek. V této práci se zaměřím především na dvě z nich: *Metodiky a techniky*.

Metodika je obecně pracovní postup (metoda) nebo nauka o metodě. Ve vývoji software metodika představuje souhrn doporučených praktik a postupů, pokrývajících celý životní cyklus vytvářené aplikace. V tomto oboru velmi často dochází – kvůli nesprávnému překladu z angličtiny – kzáměně pojmu metodika za metodologie. Pro řešení dílčích problémů mohou být v rámci nasazení metodiky uplatněny specifické postupy – metody. [20]

Technika určuje, jak se dobereme k požadovaným výsledkům, tj. určuje přesný postup kroků nebo způsob použití nástrojů, např. normalizace dat. [19]

Důvody pro zvolená hlediska jsou následující:

Metodiky. V posledních desetiletích docházelo v této oblasti k názorovým obratům ohledně efektivnosti jednotlivých metodik vývoje softwarových produktů. Pokusím se některé z metodik ohodnotit z hlediska jejich využití při odhalování softwarových chyb.

Techniky. V současné době většina středně velkých a velkých softwarových firem používá kodhalování chyb víceméně podobnou sadu standardizovaných technik, proto se na ně zaměřuji s cílem popsat jejich účinnost v procesu odhalování chyb. V posledních desetiletích se sada technik spíše vyvíjela a zdokonalovala, než aby docházelo k zásadním obratům ohledně jejich užívání.

Podle fáze, ve které byla softwarová chyba odhalena, můžeme rozlišovat chyby primární (chyba byla nalezena ve stejné fázi, ve které vznikla) a sekundární (chyba byla nalezena v pozdější fázi) [34]. Cílem odhalování chyb je maximalizovat počet chyb, které byly odhaleny ve stejné fázi, ve které vznikly.

Techniky odhalování chyb můžeme hodnotit podle jejich efektivnosti (effectiveness, podíl chyb nalezených danou technikou z celkového počtu chyb) a efektivity (efficiency, počet chyb nalezených za časovou jednotku danou technikou) [34].

V kapitole 4.2 se pokusím zhodnotit přínos metodik popsaných v kapitole 4.1 k validaci a verifikaci. Zaměřím se na fáze vývoje produktu.

V kapitole 4.4 se pokusím analyzovat příčiny chyb zmíněné v kapitole 3.1 z hlediska některých technik popsaných v kap. 4.3.

Důvodem pro tyto kombinace přístupů je jejich subjektivně nejvyšší přínos k odhalování softwarových chyb. Kombinaci metodiky a třídy chyb neshledávám příliš přínosnou, protože metodiky softwarového vývoje se k vývojovým přístupům vyslovují obecně a třídy příčin chyb závisejí spíše na jejich aplikaci na konkrétní projekt. Kombinace techniky a fáze vývoje je naopak zřejmá bez analýzy, protože techniky obvykle přímo vycházejí z fáze vývoje.

4.1 Metodiky použitelné k odhalování softwarových chyb

Metodiky týkající se softwarového vývoje přinášejí globálnější a obecnější pohled na zkoumanou problematiku. Jelikož se jedná o metodiky, tedy obecné praktiky a doporučené postupy týkající se celého

životního cyklu aplikace, je jejich zaměření obvykle obecnější, a odhalování chyb je pouze jedním z bodů, které pokrývají. V této kapitole se pokusím interpretovat jednotlivé metodiky právě ve vztahu k odhalování softwarových chyb.

Metodika vývoje softwaru je souhrn postupů, pravidel a nástrojů (frameworků) používaný pro návrh, plánování a řízení vývoje informačního systému. Metodikou se též rozumí využití takového frameworku nebo dalších specifických postupů pracovním týmem nebo celou organizací při vývoji informačního systému. [26]

V rámci metodik softwarového vývoje můžeme rozlišit dva směry. První z nich se snaží pojmout vývoj softwarového produktu chronologicky jako sled v čase navazujících kroků, jedná se o *lineárně sekvenční modely*. Druhý se zaměřuje na doporučení, jaké postupy při vývoji softwaru používat, aniž by vývoj pojímal jako lineární, jedná se o *iterativní modely*. Mezi oběma směry existuje řada přechodů. Jako iterativní lze chápat i *agilní metodiky*, v jejich rámci se však rozvinula řada dílčích modelů, proto je vyčleňuji do samostatného bodu.

1. Lineárně-sekvenční modely (nazývané též modely životního cyklu):

Model vodopád. Vodopádový model je sekvenční vývojový proces, ve kterém je vývoj nahlížen jako neustále se svažující tok fázemi analýzy požadavků, návrhu, implementace, testování (validace), integrace a údržby. Projekt je rozdělen na fáze jdoucí postupně za sebou, které se nepřekrývají, důraz je kladen na plánování, časové rozvrhy, termíny, rozpočty a realizace celého systému najednou. Přísná kontrola je udržována po celou dobu životnosti projektu prostřednictvím rozsáhlých písemných dokumentů, formálních revizí a schvalování uživatelem na konci většiny fází a vstupy od managementu před začátkem další fáze. [26] Výhodou vodopádového modelu je jeho jednoduchost k porozumění i využití, poskytuje jasnou strukturu i pro méně zkušený tým. Identifikuje milníky a časy dodání, takže projekt nespadne do časově bezbřehého vytváření produktu. Nevýhodou je, že v reálných projektech na sebe jednotlivé kroky málokdy přesně navazují, všechny požadavky obvykle nejsou jasně stanoveny hned v počátku, často dochází k prodlevám a vývoj výsledného produktu je poměrně časově zdlouhavý. Tento model není flexibilní a nepočítá s mylnými kroky ve vývoji, kvůli kterým je třeba se vracet. Je použitelný, pokud jsou požadavky na systém dobře známé, definice produktu je stabilní a technologie vývoje je hned od počátku zřejmá. Může se jednat např. o nové verze již existujícího produktu. [27][29] Variací vodopádu je model *sashimi*, u nějž se fáze mohou částečně překrývat.

V-model. V-model je upravenou verzí vodopádového modelu s důrazem na validaci (revize, inspekce) a verifikaci (testování). V každé fázi vývoje je zároveň plánováno její testování. Výhodou je, že jeho využití je jasné a jednoduché, všechny výstupy z jednotlivých fází jsou ihned prověřovány a progres lze sledovat po jednotlivých milnících. Oproti vodopádovému modelu má vyšší šanci na úspěch projektu, protože každá fáze je ihned následována testy. Nevýhodou je, že nereaguje pružně na dynamické změny a neobsahuje analýzy rizik ani nenabízí cesty pro problémy nalezené během fází testování. Je použitelný pro vývoj systémů, které vyžadují vyšší spolehlivost, v situacích, kdy jsou předem dobře známé požadavky, řešení a technologie. [28][29]

Skupina lineárně-sekvenčních metodik je nejstarší, vznikala od konce 70. let 20. století, byla široce přijímána v 80. letech a v následující dekádě podrobena kritice a postupně opouštěna. Přestože většina následujících metodik vznikala v důsledku kritické revize lineárně-sekvenčních modelů, řada z nich se o ně v určitých bodech opírala.

2. Iterativní modely:

Spirálový model. Spirálový model byl vytvořen za účelem minimalizace rizik při vodopádovém přístupu. Jednotlivé kroky [plánování – analýza – vývoj – vyhodnocení] se opakují při stale dokonalejším zvládnutí zpracovávaného problému. Výhodou je, že zahrnuje analýzu rizik, uživatel má brzy možnost vidět prototyp produktu a dát zpětnou vazbu, a návrh ani první verze produktu nemusí být dokonalá. Nevýhodou je jistá časová a finanční ztráta způsobená analýzou rizik (u malých nebo málo rizikových projektů nemusí být analýza rizik nutná), celkově je tento přístup časově poměrně náročný a časově těžko ohraničitelný a může být obtížné definovat konkrétní a ověřitelné milníky. Model je použitelný pro vývoj zcela nových produktů, pro velké systémy a projekty, kde je vysoká míra rizika nebo je analýza rizik důležitá. [28] Aplikací spirálového modelu na objektové technologie je model skládání komponent. Podtypem tohoto modelu je také win-win spirálový model, který klade důraz na jasné specifikace cílů, omezení a alternativ z hlediska jednotlivých zainteresovaných subjektů.

Iterativní model. Je kombinací sekvenčních a inkrementálních metodik. Celý vývoj se odehrává v několika opakováních, které směřují k postupnému vylepšování, zpřesňování, dodělávání a opravení systému. Výsledkem je v každé iteraci větší a lépe fungující systém. Každá iterace obsahuje miniaturní vodopád [návrh – analýzu – vývoj – testování], ale v každé iteraci se klade důraz na jinou část vývoje (v první iteraci se např. provede detailní návrh a rozpracuje se jádro systému a návrh implementace, ve druhé se dodělá návrh a rozpracují důležité části systému a pokusí se o částečnou implementaci atd.). Výhodou jsou brzy dostupné dílčí výsledky a časné odhalení chyb. Nevýhodou je dosti dlouhá celková doba vývoje systému a nutnost neustálého předělávání kódu (je sporné, zda je toto nevýhoda). [28]

Inkrementální (přírůstkový) model je kombinací sekvenčních a iterativních metodik. Cílem je omezit projektová rizika rozdělením projektu na menší segmenty a zjednodušení procesu změn uprostřed vývoje. Jednotlivé „přírůstky“ (inkrementy) se vytvářejí nezávisle na zbytku systému a jsou pak integrovány. Na rozdíl od přístupu RAD jednotlivé přírůstky procházejí samostatným testováním, implementací a údržbou. Výhodou je možný paralelní vývoj ve více týmech, možnost vývoje v malých týmech, možnost realizovat jednotlivé přírůstky různými metodami (vodopádem, iterativně, agilně) a snadná modifikace a flexibilita. Nevýhodou je nutnost kvalitního počátečního plánování a návrhu včetně úplné definice plně funkčního systému rozdělitelného na inkrementy, dále nutnost dobře definovat rozhraní začleněných modulů. Inkrementální přístup je vhodný pro dlouhodobé projekty, které nesmějí selhat. [27] Příkladem inkrementálního modelu je Software Development Life Cycle (SDLC).

Objektově orientovaný model: Rapid Application Development (RAD). Jedná se o přechod mezi lineárním a iterativním prototypovým modelem. Po fázi analýzy a návrhu následuje iterativní fáze vývoje prototypu [stavba – předvedení – zlepšení], následovaná testováním a implementací. Výhodou je poměrně rychlý vývoj a relativně dobrá efektivita vzhledem k vynaloženým prostředkům. Problém se rozdělí na několik komponent, které se přiřadí různým týmům. Nevýhodou je chaotické, místy živelné narůstání produktu, které se obtížně dokumentuje a dohledává. Proto je přístup vhodný pro menší projekty s kratším obdobím vývoje. [26][29]

Prototypové modely se zaměřují na vývoj prototypu softwaru, který nemusí být úplný. Jedná se spíše o postup k částem produktu v rámci větších vývojových přístupů než o samostatnou metodiku. Cílem je snížení projektových rizik rozdělením produktu na menší části a cílem je pochopení požadavků na systémy, které nelze specifikovat předem. U většiny prototypů je vyvíjena jen určitá vlastnost a ne celková funkcionalita. Nevýhodou je tendence sklouznout k chaotickému programování a neustálému

opravování. Použitelné jsou pro menší systémy s nejasnými požadavky, nové a originální návrhy, rychlé demonstrace systému a vývoj uživatelského rozhraní. [26]

Rational Unified Process (RUP) aplikuje iterativně-inkrementální vývoj. Nejde o konkrétní normativní model, spíše o adaptabilní rámec vývoje. Produkt prochází přes čtyři fáze: zahájení – rozpracování – konstrukce – předání. Každá fáze může být rozpracována do různých iterací (cyklů) a každý cyklus obsahuje plánování – analýzu – návrh – implementaci – testování – novou verzi systému. [27]

3. Agilní metodiky

Agilní metodiky se snaží o rychlejší a levnější vývoj produktu, než umožňují předchozí metodiky. Zdůrazňují jako primární měřítko progresu fungující software a dodržování uzávěrek. Většina těchto metodik se pokouší o minimalizaci rizik pomocí vývoje v krátkých iteracích trvajících 1-4 týdny (timebox). Každá iterace funguje jako samostatný projekt a zahrnuje všechny úkoly nutné ke zveřejnění nové funkcionality [plánování – požadavky – návrh – kódování – testování – dokumentaci]. Pokud není nová funkcionalita na konci iterace dokončena, je tým přesto schopen uvolnit novou verzi produktu, byť s omezenou funkcionalitou.

Tyto metodiky upřednostňují komunikaci tváří v tvář před psanými dokumenty, ve srovnání s jinými metodikami vytvářejí mnohem méně dokumentace. Většina agilních týmů je soustředěna na jednom místě, nejlépe v jedné místnosti, a zahrnuje všechny členy týmu nutné k provedení kompletní iterace. Výhodou je obvykle levnější i rychlejší vývoj produktu, nevýhodou může být neustálé hektické nasazení všech členů týmu a problematické stanovení priorit změn. Členové týmu musí mít dostatek zkušeností, aby nesklouzli k chaotickému programování a následnému opravování chyb. Agilní metodiky jsou vhodné pro malé a středně velké projekty, nikoliv pro velké projekty, kde je klíčová dokumentace. [27][30]

Agilní metodiky zahrnují řadu specifických přístupů, které se mohou kombinovat vzájemně i s jinými metodikami, např.:

Extrémní programování (XP) se soustředí na vývoj a dodávání velmi malých inkrementů funkcionalit. Zdůrazňuje neustálé vylepšování kódu (včetně refaktorizace), zahrnutí uživatelů do vývojářských týmů a párové programování. Všichni vývojáři pracují na všech oblastech kódu, nevytváří se expertiza. Velký důraz je kladen na testy.

Crystal metodiky. Důraz je kladen na talent a schopnosti členů týmu, interakci a komunikaci. Předpokládá se, že tyto vlastnosti jsou pro úspěch projektu důležitější než obecné procesy; naopak procesy by měly být přizpůsobeny vlastnostem konkrétního týmu.

Dynamic Systems Development Model (DSDM). Je založen na metodikách RAD. Důraz je kladen na zplnomocnění týmů činit rozhodnutí, časté vydávání nových verzí produktu, schopnost produktu plnit obchodní účely, vratné změny během vývoje, integrační testování, spolupráci všech zainteresovaných subjektů.

Feature Driven Development (FDD). Důraz je kladen na jednoduché, dobře definované procesy, srozumitelné všem členům týmu. Dobře definované procesy tvoří pevný základ, takže se členové týmu mohou soustředit na výsledky. Obvyklý zastřešující proces má následující prvky: vývoj celkového modelu – vytvoření seznamu vlastností produktu – plán podle vlastností – návrh podle vlastností – stavba podle vlastností.

Test Driven Development (TDD). Tento přístup ve zvýšené míře reflektuje důraz agilních metodik na testování. Někdy bývá začleňován jako jedna z technik extrémního programování, jindy vyčleňován jako samostatný metodický přístup. Při TDD se využívá automatických unit testů testujících nejmenší jednotku programu, které se opětovně spouštějí. Prvním krokem při vývoji podle TDD je vytvoření testu ověřujícího očekávanou funkcionalitu, dále jejich spuštění (očekává se, že při prvním spuštění všechny testy neprojdou), kódování, opětovné spouštění testu dokud všechny neprojdou, a refaktorizace. Tento cyklus se opakuje, dokud nejsou všechny funkcionality úspěšně implementovány. [25]

Lean Development (LD). Zaměření tohoto přístupu je především na produkt dobře snášející změny – vykazující dynamickou stabilitu. Velký důraz je kladen na spokojenost zákazníka, velkou motivovanost vývojářů, minimalismus, týmové úsilí.

Joint Application Development (JAD). Tato metodika se zaměřuje na definici požadavků a uživatelský interface, tedy na části vývoje, v nichž je klíčová komunikace s koncovými uživateli produktu. Jedná se o metodiku soustředěnou spíše na obchodní než technickou stránku problému, a to pomocí řízených sestav workshopů (JAD sessions).

Scrum. Tato metodika je určena pro řízení projektu a cílem je intenzivně zvýšit produktivitu týmů svázaných metodikami orientovanými na procesy. Základem jsou stručné denní mítinky (scrums), kde jsou určeny priority zbývající práce a rozděleny do krátkých pracovních iterací (sprints).

Při předchozím popisu agilních metodik se opírám především o literaturu [30]. V rámci metodik softwarového vývoje by bylo možné jmenovat řadu dalších, v této práci jsem se omezila jen na některé.

4.2 Přínos metodik softwarového vývoje k odhalování chyb

Pokud vezmeme v potaz, že jedním z hlavních cílů validace a verifikace je co nejčasnější nalezení chyb, pak *vodopádový model* toto kritérium zřetelně nesplňuje, protože fáze testování se nalézá až na konci vývojového cyklu produktu.

Oproti tomu *V-model* akcentuje testování v každé fázi vývoje, takže maximalizuje počet primárních chyb oproti sekundárním, a v tom je jeho výhoda. Problém nastává, pokud během projektu dochází ke změnám, nebo se určitý vývojový směr ukáže jako nesprávný (např. pokud se na konci fáze kódování ukáže, že je část konceptu nevhodná a je nutné provést změny návrhu). Navíc je málo pravděpodobné dodatečné odhalení chyb vzniklých v předchozí fázi vývoje, protože se testuje pouze aktuální vývojová fáze.

Spirálový vývoj podobně jako V-model zahrnuje verifikaci všech vývojových fází. Specifikace požadavků i návrh jsou následovány jejich testováním, následuje plán integrace a testování a po uvolnění celkového produktu sada testů jednotkových, integračních i systémových. Z pohledu verifikace je spirálový model vhodný, problém je v jeho časové zdlouhavosti. Problematické může být také předcházení plánu testování před detailním návrhem, protože při plánování nemusí být dořešeno, jak konkrétně budou vlastnosti produktu implementovány, a jak přesně je tedy testovat. Plán testů tedy není definitivní a měl by být flexibilní ke změnám vzniklým v poslední fázi návrhu a implementace.

V rámci *iterativního vývoje* probíhá validace opakovaně na konci každé iterace, v určité fázi vývoje se k iteracím přidají akceptační testy (nejprve probíhají ve vývojářské firmě, posléze u zákazníka). Výhodou je velká šance odhalení primárních chyb vzniklých ve fázi kódování díky časné a časté validaci.

Jako nedostatek lze hodnotit, že iterativní vývoj explicitně neuvádí verifikaci specifikací a návrhu. Naopak přínosné je zahrnutí testů na straně zákazníka do základního modelu.

Inkrementální model verifikaci nezdůrazňuje, testování je zakončením stavby systému na konci vývoje jednotlivých inkrementů. Výhodou tohoto modelu je flexibilita alokace lidských zdrojů, tedy programátorského i testovacího týmu; testerské týmy mohou např. postupně testovat jednotlivé inkrementy a ve vývoji nedochází k časovým prodlevám, nebo na různých inkrementech mohou pracovat různé týmy. Nevýhodou tohoto modelu je nutnost vynikající organizace verzí jednotlivých inkrementů, potažmo posléze celého systému. Při použití tohoto modelu je vhodnější postupná integrace inkrementů s následným testováním celku než oddělený vývoj a testování samostatných inkrementů s integrací v poslední fázi.

U *Rapid Application Developmentu* testování probíhá na konci vývoje, což vzhledem k určení modelu (rychlý vývoj nepříliš rozsáhlých aplikací) nepředstavuje závažný problém. Problém s tímto může vzniknout, pokud je RAD nevhodně použit na rozsáhlejší systémy s delší dobou vývoje nebo pokud je vývoj příliš chaotický, což se u RAD občas může přihodit.

Prototypové modely testování obvykle nezdůrazňují, je součástí vývoje každého prototypu. Ohledně validace jednotlivých prototypů lze při užití vhodných technik předpokládat přiměřenou úspěšnost detekce chyb, problémem může být testování celého systému vzniklého sloučením prototypů.

Agilní metodiky obvykle k testování přistupují jako k důležitému článku vývoje, mj. díky jejich vyzdvihování nutnosti dodat fungující verzi software v dohodnutém čase. Testování je většinou dobře rozpracováno, snahou je testovat co největší množství scénářů v co nejpodrobnějších sadách testů, důraz je kladen i na programátorské testování (unit testy jsou psány vždy před začátkem kódování). Při nezvládnutí organizace vývoje jednotlivých verzí může mít časový tlak na testování negativní vliv; může docházet k posunům začátku testování jako poslední fáze před uvolněním produktu, aniž by se posouvaly uzávěrky, což v kombinaci se snahou o maximalizaci počtu testů může vést k nekvalitnímu testování a značnému stresu testerského týmu s brzkým vyčerpáním psychických rezerv.

Z hlediska testování má v rámci agilních metodik zvláštní postavení *Test Driven Development* (TDD). Důraz na testování je zde doveden do extrému, psaní automatizovaných jednotkových testů předchází kódování a při jakýchkoliv změnách jsou tyto sady testů spouštěny neustále dokola. V mnoha případech jde víceméně o neustálé automatizované regrese, které při vhodném použití přispívají k udržení stability již vytvořeného kódu. Nevýhodou je, že testy značně zvyšují objem kódu, který je třeba napsat (objem testovacího kódu může dokonce převyšovat objem kódu testovaného), a skutečnost, že v testech se stejně jako v jiných částech kódu mohou vyskytovat chyby. Tento přístup nemůže nahradit testování celku, protože jednotkové testy jsou slabým nástrojem k odhalení komplikovanějších chyb.

Tab. 7 Výhody a nevýhody metodik softwarového vývoje z hlediska validace a verifikace.

Metodika	Výhody	Nevýhody
vodopádový model	jasná posloupnost verifikačních postupů	pozdní verifikace
V-model	včasná a robustní verifikace	malá flexibilita
spirálový model	včasná verifikace	časová zdlouhavost, plánování testů předchází detailnímu návrhu
iterativní modely	časná a častá validace, zahrnutí akceptačních testů	nepracuje s verifikací specifikace a návrhu
inkrementální modely	flexibilita lidských zdrojů, nedochází k časovým oknům	nutnost precizní organizace verzí a systému
RAD	vhodný pro kratší projekty	verifikace na konci
prototypové modely	časná verifikace prototypů	riziko při verifikaci sloučených prototypů
agilní metodiky	detailně rozpracovaná validace i verifikace	časový stres
TDD	zdůrazněná validace	příliš zdůrazněná validace, nenahrazuje testování celku

4.3 Techniky odhalování softwarových chyb

Mezi techniky odhalování chyb během softwarového vývoje řadíme např.

- Revize a inspekce návrhu
- Revize a inspekce kódu
- Modelování a prototypování
- Osobní ruční kontrola kódu
- Testování jednotek a komponent
- Integrační, regresní a systémové testování
- Beta-testování

McConnell uvádí efektivnost jednotlivých technik pomocí procentuálního vyjádření nalezených chyb z celkového počtu chyb v dané fázi projektu:

Tab. 8 Procento chyb nalezených různými technikami. [16, s. 484]

Neformální revize návrhu	35%
Formální inspekce návrhu	55%
Neformální revize kódu	25%
Formální inspekce kódu	60%
Modelování nebo prototypování	65%
Osobní ruční kontrola kódu	40%
Testování jednotek	30%
Testování nové funkce (komponenty)	30%
Test integrace	35%
Regresní testování	25%
Test systému	40%
Menší beta-testování (<10 pracovišť)	35%
Rozsáhlé beta-testování (>1000 pracovišť)	75%

Vzhledem k tomu, že průměrná úspěšnost žádné z technik nepřesáhla 75% a obvykle se pohybovala kolem 40%, je naprosto nezbytné kombinovat různé techniky odhalování chyb. Různorodost odhalovaných chyb je tak velká, že jakákoliv kombinace dvou metod zvýší celkový počet nalezených chyb téměř na dvojnásobek. Dále se ukazuje, že různí lidé nalézají různé chyby, takže je vhodné, aby jednu techniku provádělo více lidí. [16] V praxi je tak vhodnější, aby např. revize a inspekce kódu prováděli různí lidé, nebo aby se tým testerů cyklicky střídal v integračním, regresním a systémovém testování, pokud to jejich znalosti umožní.

Ukazuje se, že pro vyhledávání určitých typů chyb jsou úspěšnější postupy prováděné ručně, pro jiné typy chyb je vhodnější počítačové testování. „Kumulativní efektivita při detekci chyb je výrazně vyšší než jakákoliv individuální technika. Vyhlídka, že izolované testování bude efektivní, je tristní. […] Kombinace testování jednotek, funkčního testování a testování systému obvykle vyústí v kumulativní odhalení méně než 60% chyb, což je obvykle pro ostrý software nedostatečné.” [16, s. 485]

Z hlediska nákladů na nalézání a odstraňování chyb jsou však kumulativní kombinace technik dražší než jednotlivé techniky, proto je třeba zvážit, které techniky hledání chyb do vývoje zahrnout. McConnell navrhuje doporučenou kombinaci vhodnou k dosažení vyšší než průměrné kvality: [16, s. 487]

- neformální revize všech požadavků, celé architektury a celého návrhu nejdůležitějších částí systému,
- modelování a prototypování,
- čtení kódu a přezkoumání kódu
- testování běhu.

Průměrná aktivita spojená se softwarovým produktem je přibližně 10-50 řádků kódu dodaných jednou osobou za den. Průměrné hodnoty sice zohledňují i neprogramátorské pracovníky, nicméně je zřejmé, že hlavní část pracovního dne programátorů nezabírá programování nového kódu, ale ladění, opravy a refaktorizace kódu. Podle McConnella je řešením investice do aktivit probíhajících v časných fázích projektu, protože mají na kvalitu software větší vliv než aktivity probíhající v pozdějších fázích. Čas investovaný do projektu ve fázi požadavků a návrhu sníží množství chyb v dalších fázích projektu a tím zkrátí dobu vývoje a ušetří náklady na vývoj. [16, s. 489]

Z hlediska množství chyb v kódu je průmyslový průměr kolem 1-25 chyb na 1000 řádků kódu dodaného softwaru. Ke snížení této hodnoty lze použít cílené kombinace technik. McConnell [16, s. 530] uvádí, že:

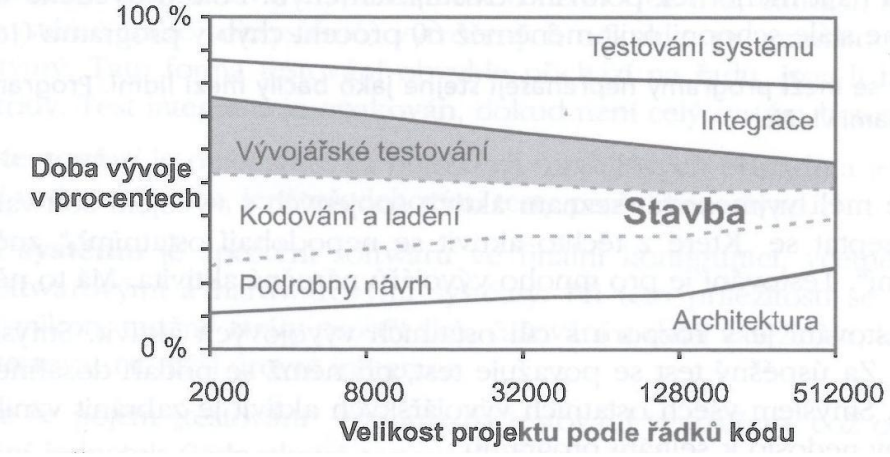
- Microsoft dosáhl hodnoty 0,5 chyb na 1000 řádků kódu v uvolněném produktu pomocí kombinace společných technik vývoje (párové programování, formální inspekce, neformální revize kódu).

- Pomocí techniky „vývoje v čisté místnosti“ bylo dosaženo 0,1 chyb na 1000 řádků v uvolněném produktu. Takový výsledek byl dosažen využitím kombinace formálních vývojových metod, přezkoumávání kódu kolegy a statistickým testováním.

- Týmy používající metodiku TSP (Team Software Process) dosáhly podílu kolem 0,06 chyb na 1000 řádků kódu v uvolněném produktu. Tato metoda se zaměřuje na školení vývojářů, v němž se snaží, aby primárně nedělali chyby.

Tentýž autor se zabývá rovněž návrhem časové struktury vývoje projektu, a to z hlediska velikosti projektu. Dochází k závěru, že s rostoucí velikostí projektu by měl klesat podíl stavebních aktivit (návrh, kódování a ladění,vývojářské testování) v celkovém času projektu:

Graf 4 Procento času strávené aktivitami v závislosti na velikosti projektu. [16, s. 512]



4.4 Přínos technik softwarového vývoje k odhalování chyb

Téma vyhledávání a odhalování chyb v softwaru pomocí různých technik je častým námětem odborných studií. Určitá část se soustředí na porovnávání dvou či více technik odhalování chyb. Existuje řada studií, které se pokoušejí o porovnávání efektivnosti inspekce kódu a testování v nalézání chyb, nicméně výsledky jsou značně nekonzistentní v závislosti na konkrétním projektu a zvolené metodě porovnávání. Další studie se zaměřují na kumulativní efekty různých technik při odhalování chyb. Některé zkoumají rovněž účinnost různých způsobů testování ve vztahu k různým příčinám chyb. Velkým problémem týkajícím se zejména poslední skupiny uvedených studií je skutečnost, že klasifikace i způsoby verifikace obvykle odpovídají potřebám konkrétního projektu a jsou jen těžko zobecnitelné (např. jako příčina se hodnotí mylné použití < namísto > nebo =<, špatná poloha senzoru v důsledku nejednotnosti specifikací, jako porovnání technik funkční vs. náhodné testování, certifikační vs. operační testování, back-to-back testing vs. schvalování více osobami). Řada studií byla provedena laboratorním způsobem (odhalování uměle vytvořených chyb apod.), takže jejich validita je sporná.

Při vyhledávání statí na toto téma zainteresovaný člověk nejprve získá dojem, že téma klasifikace příčin softwarových chyb ve vztahu k technikám odhalování chyb je již zcela vyčerpáno, při jejich procházení naopak dojde k přesvědčení, že existuje nesčetná řada fragmentů pokrývající tuto oblast, nikoliv však jednotné, prakticky obecně použitelné závěry. To může být způsobeno skutečností, že zkoumání této oblasti je otázkou teprve posledního desetiletí, avšak také faktem, že k jednotným závěrům nelze dospět, pokud člověk nechce sklouznout k banálním konstatováním. Jak uvádí [35], existující důkazy jasně neukazují přednost určité inspekční techniky nad jinou, určité testovací techniky nad jinou, nebo určité inspekční techniky nad určitou testovací technikou, nebo vzájemnou korelaci různých technik. Tentýž zdroj uvádí, že podle některých studií mají inspekce vyšší efektivnost (počet chyb nalezených za časovou jednotku) než testování, ale testování má vyšší efektivitu (podíl nalezených chyb z celkového počtu chyb) než inspekce, podle jiných studií mají inspekce, funkční testování a strukturální testování podobnou efektivnost. Inspekce kódu jsou efektivnější při nalézání chyb, testování při nalézání

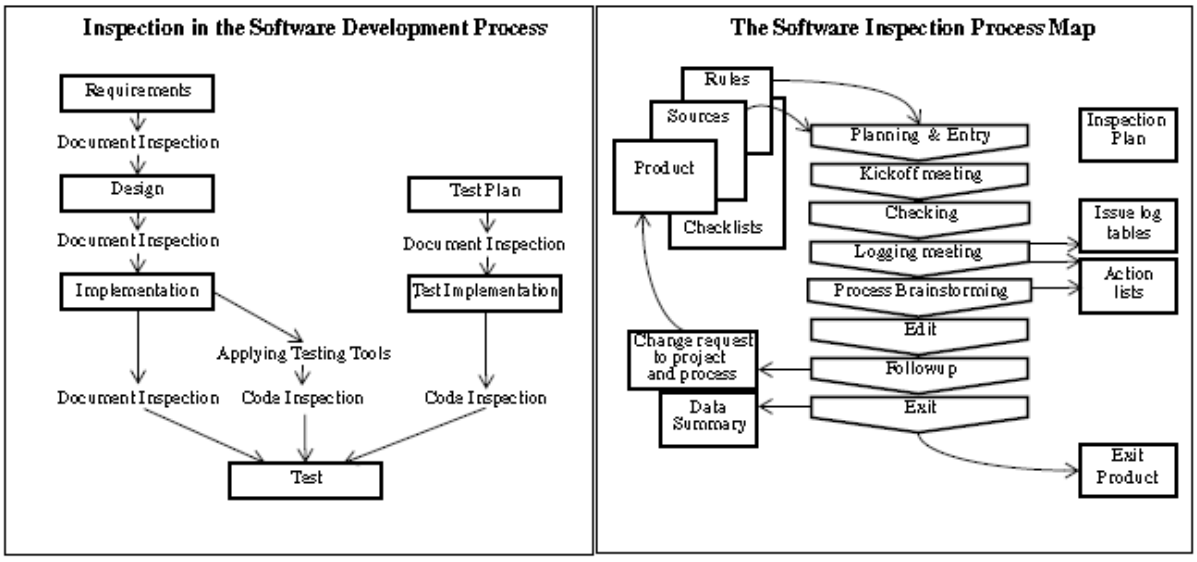
chybějících funkcionalit [34]. Podle studie [33], porovnávající úspěšnost různých technik detekce chyb při revizi specifikace požadavků, měly srovnatelnou úspěšnost metoda kontrolního seznamu (checklist) a dodatečná detekce (ad hoc detection), o více než polovinu vyšší úspěšnost měla revize scénářů. Podle [35] jsou rozdíly v úspěšnosti těchto technik (obecně, bez vztahu ke specifikacím) zanedbatelné.

Je zřejmé, že existující výzkum této oblasti prozatím nenabízí kompletní uspokojivé řešení. Porovnávání technik ve smyslu „která je lepší“ nepřináší užitečná řešení, spíše má význam uvažovat ve smyslu „která je lepší k čemu“. Pokusím se shrnout dosavadní poznatky ohledně technik softwarového vývoje s ohledem na potřeby zadavatelské firmy této práce.

Review softwaru. Tato technika se vztahuje k review požadavků, návrhu i zdrojového kódu. Obvykle je méně formální než inspekce a používá méně rigidní postupy. Obvykle jde o formu prezentace autorů požadavků (návrhu, komponenty) následovanou otevřenou diskuzí účastníků a shrnutím ze strany vedoucího projektu. Výsledky review jsou zpřístupněny všem členům týmu. Review se zaměřuje na čitelnost, jasnou objektivitu, flexibilitu vůči případným budoucím změnám, porovnání alternativních návrhů, stav jednotkových testů apod. Cílem je *rozhodnout o další fázi projektu*. Organizátoři review musí mít dostatek zkušeností (alespoň 3 roky práce na podobných projektech, po 5 letech zkušeností už přínos dále nevzrůstá). [36][37] Některé studie uvádějí vysokou úspěšnost review kódu ve vztahu k hledání chyb, takové studie však obvykle posuzují review prováděné na nezkompilovaném kódu. Vzhledem ke kompilovanému software je úspěšnost této techniky nižší.

Inspekce softwaru. Tato technika je typem formálního review aplikovatelného na jakýkoliv typ artefaktu, používá jasně definované vstupní a výstupní požadavky, rozdělení rolí účastníků, měřící činnosti. Cílem je *identifikace a oprava chyb* (na rozdíl od jiných typů review). Softwarová inspekce je schopna odhalit přibližně polovinu chyb v kódu, opakovaná inspekce odhalí polovinu zbývajících chyb. Nejlepší výsledky poskytuje inspekce prováděná ve dvou lidech (ne méně, ne více). Jednou z úspěšných inspekčních technik je čtení vycházející z chyb (defect-based reading). Při této technice se vytvoří scénář založený na chybách a k němu detailní sada instrukcí, podle nichž se provádí inspekce. Efektivnější než inspekční meetingy jsou inspekce prováděné 1-2 kvalifikovanými osobami. Inspekce by měla probíhat pouze na hotových částech kódu, měla by trvat maximálně dvě hodiny a probíhat rychlostí asi 6 stran za hodinu. Výsledkem inspekce musí být sesbírané záznamy o chybách. [35]

Obr. 3 Proces inspekce softwarového vývoje a jeho mapa [37]



Modelování a prototypování. Jde o proces vytvoření neúplného modelu budoucího funkčního software. Cílem je *ověřit realizovatelnost stávajícího návrhu*. Tato technika může vizualizovat navrhnuté řešení a tím pomoci doplnit specifikace požadavků, přispět k prevenci odchylek vývojářů a softwarových inženýrů od původního návrhu, zpřesňovat další postupy a stanovovat dosažitelné uzávěrky. Je vhodným způsobem zvýšení participace zákazníků na projektu a jejich zpětné vazby. Ukazuje se, že vhodné využití modelů a prototypů ve fázi specifikace a návrhu může významným způsobem ušetřit až 40% času a nákladů ve srovnání s postupem bez prototypů, protože snižuje množství času stráveného vývojem produktu a testy. [22] Tuto techniku zdůrazňuje prototypový model.

Osobní ruční kontrola. Tato technika se může vztahovat jak ke kontrole zdrojového kódu, tak také ke kontrole organizace (verzí, komponent apod.). V případě kontroly zdrojového kódu se jedná o součást ladění programu. Cílem je *rozpoznat příčiny chyby a následně je opravit*. Efektivní postup spočívá v nalezení chyby, nalezení její příčiny, opravení, prověření opravy a snaze nalézt další podobné chyby. Neefektivní postup spočívá v náhodných, chaotických změnách ve snaze zprovoznit program bez nalezení příčiny chyby. Ve srovnání s programátorským testováním je ruční kontrola kódu poměrně málo efektivní. Velmi důležitá je však při důsledné kontrole organizace verzí a komponent, protože řada tzv. opakujících se chyb vzniká právě v jejich důsledku, kdy se např. oprava dostane do testovací verze, avšak nikoliv do ostré. [16]

Testování jednotek a komponent. Testování jednotek spočívá ve spuštění hotové třídy, rutiny nebo malého programu, jenž byl napsán jedním programátorem nebo týmem programátorů. Testování komponent spočívá ve spuštění třídy, balíčku, malého programu nebo jiného programového prvku, který je výsledkem činnosti několika programátorů nebo vývojových týmů. Tyto moduly jsou testovány nezávisle na zbývající části systému. Vývojářské testování by mělo pokrývat všechny důležité požadavky a důležité aspekty návrhu. Všechny chyby by měly být uvedeny v kontrolním seznamu. Vývojářské testy by měly obsahovat testy splnění i testy selhání. Častěji prověřovány by měly být třídy a rutiny, ve kterých už se v minulosti chyby vyskytly, a často se měnící části kódu. Technikou programátorského testování lze odhalit 30-80% chyb, obvykle asi 50%. [16] Tuto techniku zdůrazňují agilní metody vývoje. Velmi účinnou, avšak náročnou technikou programátorského testování je testování všech možných cest programu (path testing).

Integrační, regresní a systémové testování. Cílem integračního testování je ověřování, že nově přidané funkcionality spolu nekolidují a pracují správně i po integraci do hlavního projektu. Ověřuje se funkčnost, výkon a spolehlivost. Existuje několik typů integračního testování, které se odlišují postupem, např. lze testovat integraci hotového softwaru bez předchozích dílčích integračních testů, postupovat od komponent na nižších úrovních ke komponentám na vyšší úrovni nebo obráceně. Cílem systémového testování je ověřit, že po přidání nových funkcionalit celý systém pracuje správně a odpovídá všem požadavkům. Systémové testy jsou obvykle značně rozsáhlé, vedle samotné funkcionality ověřují např. také uživatelské rozhraní, výkon, kompatibilitu, fungování při zátěži, bezpečnost, různé konfigurace. Součástí systémových testů jsou regresní testy, které ověřují, zda nově přidané funkcionality neporušily funkcionality původní. Obvyklým postupem regresních testů je opakované spouštění regresní sady testů.

Beta-testování. Jedná se o proces externího testování, při němž se software zašle určité vybrané skupině potenciálních zákazníků, kteří s ním budou pracovat na reálném prostředí. Beta-testování probíhá obvykle ke konci vývojového cyklu produktu. Cílem beta-testování je ověření reálné činnosti systému, kompatibility, konfigurace a použitelnosti. Mimo tyto oblasti je beta-testování poměrně neúčinnou metodou hledání chyb, nicméně bývá cennou metodou získání solidních a nezávislých údajů o testování softwaru. [8]

V následujících tabulkách (Tab. 9 – Tab. 13) jsem se pokusila o návrh vhodných technik odhalování chyb ve vztahu k příčinám chyb z hlediska jednotlivých fází vývoje softwaru. Vycházím z výše uvedených poznatků a tabulek použitých v kap. 3.1 (Tab. 1 – Tab. 6).

Tab. 9 Návrh technik a faktorů odhalování chyb ve vztahu k příčinám chyb a specifikaci

Třída chyb	Specifikace požadavků
Komunikace	review specifikace
Účast	zákaznický review specifikace (účast autora specifikace a zákazníka)
Znalost problematiky	zkušenosti autora specifikace
Specifická znalost aplikace	architektův review specifikace (účast autora specifikace a sw architekta)
Průběh procesu	architektův review specifikace (účast autora specifikace a sw architekta)
Jiné znalosti	zkušenosti autora specifikace
Nevhodná metoda dosahování cílů	zkušenosti autora specifikace
Řízení	zkušenosti manažerů
Předávání znalostí	modelování a prototypování (účast vedoucích projektu a programátorů)
Analýza	architektův review specifikace (účast autora specifikace a sw architekta)
Sledovatelnost	osobní ruční kontrola specifikace a návrhu (sw architekt)
Organizace	použití vhodných nástrojů záznamu dokumentace
Absence standardů	vytvoření standardů
Specifikace	zákaznický review specifikace (účast autora specifikace a zákazníka)

Tab. 10 Návrh technik a faktorů odhalování chyb ve vztahu k příčinám chyb a návrhu

Třída chyb	Návrh a implementace
Komunikace	review návrhu
Účast	review návrhu (účast autora specifikace, sw architekta, vedoucích projektu)
Znalost problematiky	zkušenosti sw architekta
Specifická znalost aplikace	review návrhu (účast sw architekta, vedoucích projektu)
Průběh procesu	review návrhu (účast sw architekta, vedoucích projektu)
Jiné znalosti	review návrhu (účast sw architekta, vedoucích projektu)
Nevhodná metoda dosahování cílů	zkušenosti sw architekta
Řízení	zkušenosti manažerů
Předávání znalostí	zkušenosti sw architekta
Analýza	zkušenosti sw architekta
Sledovatelnost	osobní ruční kontrola návrhu a částí programu (vedoucí projektu)
Organizace	zkušenosti sw architekta
Absence standardů	vytvoření standardů
Specifikace	review návrhu (účast autora specifikace, sw architekta, vedoucích projektu)

Tab. 11 Návrh technik a faktorů odhalování chyb ve vztahu k příčinám chyb a programování

Třída chyb	Programování (v tomto sloupci dokument = program)
Komunikace	modelování a prototypování
Účast	modelování a prototypování (účast vedoucích projektu a programátorů)
Znalost problematiky	zkušenosti vedoucích projektu
Specifická znalost aplikace	zkušenosti některých programátorů + předávání know-how ostatním programátorům
Průběh procesu	modelování a prototypování (účast vedoucích projektu a programátorů)
Jiné znalosti	inspekce kódu
Nevhodná metoda dosahování cílů	inspekce kódu
Řízení	zkušenosti manažerů
Předávání znalostí	inspekce kódu
Analýza	inspekce kódu
Sledovatelnost	použití vhodných nástrojů záznamu komponent a verzí
Organizace	zkušenosti některých programátorů + předávání know-how ostatním programátorům
Absence standardů	zkušenosti některých programátorů + předávání know-how ostatním programátorům
Specifikace	inspekce kódu

Tab. 12 Návrh technik a faktorů odhalování chyb ve vztahu k příčinám chyb a testování

Třída chyb	Testování (v tomto sloupci dokument = testovací scénář nebo testovací sada)
Komunikace	modelování a prototypování
Účast	modelování a prototypování (účast vedoucích projektů a testerů)
Znalost problematiky	zkušenosti autora testů
Specifická znalost aplikace	zkušenosti některých testerů + předávání know-how ostatním testerům
Průběh procesu	review plánovaných testů (účast vedoucích projektu a autora testů)
Jiné znalosti	zkušenosti některých testerů + předávání know-how ostatním testerům
Nevhodná metoda dosahování cílů	review plánovaných testů (účast vedoucích projektu a autora testů)
Řízení	zkušenosti manažerů
Předávání znalostí	modelování a prototypování (účast vedoucích projektů a testerů)
Analýza	review plánovaných testů (účast vedoucích projektu a autora testů)
Sledovatelnost	použití vhodných nástrojů záznamu průběhu testů
Organizace	použití vhodných nástrojů záznamu průběhu testů
Absence standardů	použití vhodných nástrojů záznamu průběhu testů
Specifikace	review plánovaných testů (účast vedoucích projektu a autora testů)

Tab. 13 Návrh technik a faktorů odhalování chyb ve vztahu k příčinám chyb a údržbě

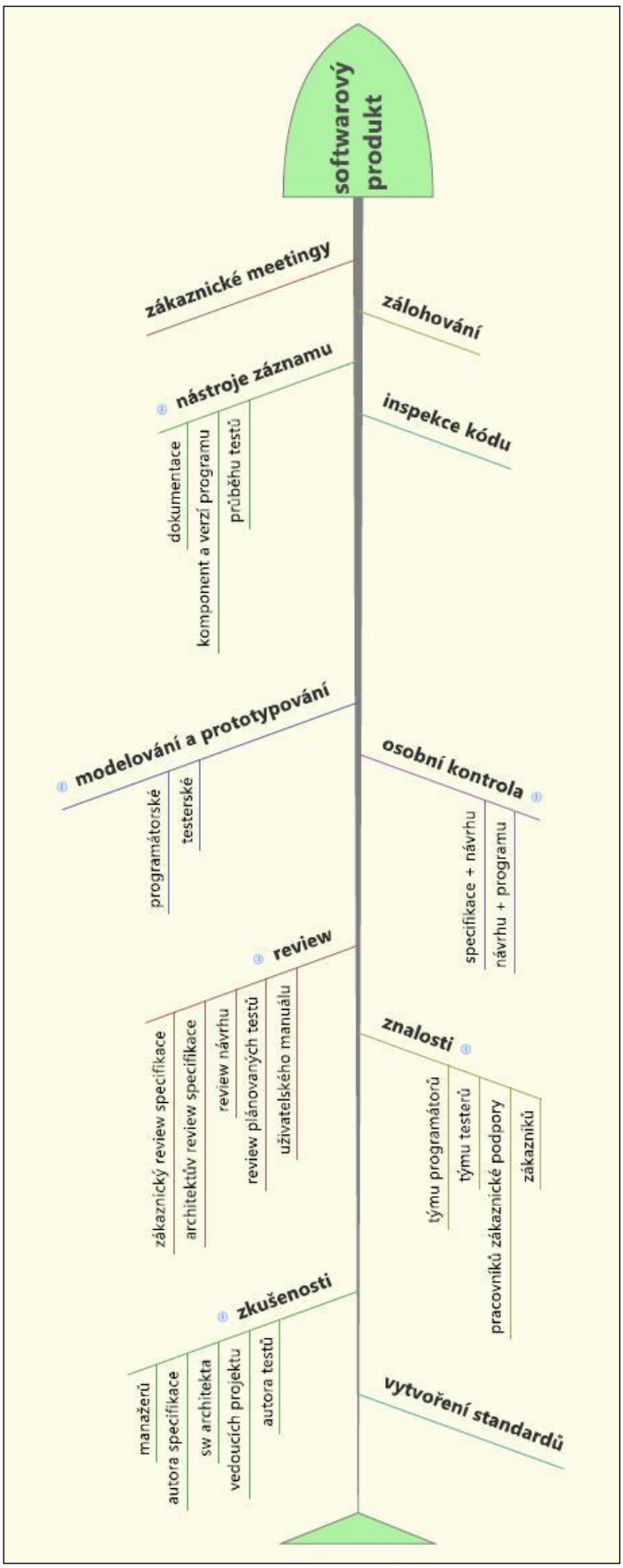
Třída chyb	Údržba
Komunikace	pravidelné meetingy
Účast	pravidelné meetingy (účast vedoucích projektu a zákazníka)
Znalost problematiky	proškolení pracovníků zákaznické podpory
Specifická znalost aplikace	proškolení pracovníků zákaznické podpory
Průběh procesu	pravidelné meetingy (účast vedoucích projektu a zákazníka)
Jiné znalosti	proškolení pracovníků zákaznické podpory
Nevhodná metoda dosahování cílů	pravidelné meetingy
Řízení	pravidelné meetingy
Předávání znalostí	proškolení zákazníků
Analýza	pravidelné meetingy
Sledovatelnost	zálohování
Organizace	zkušenosti manažerů
Absence standardů	review uživatelského manuálu
Specifikace	review uživatelského manuálu

Na základě předchozích tabulek jsem vytvořila diagram příčin a následků (Graf č. 4). Je zřejmé, že pro případné praktické využití by tento graf vyžadoval další rozpracování jednotlivých kroků, kritické posouzení jejich přínosu a zhodnocení finančních aspektů jejich aplikace. Tato analýza by však již překročila rozsah této práce, proto ji ponechávám jako otevřenou otázku do budoucnosti.

Graf 5 Návrh technik a klíčových faktorů při vytváření softwarového produktu z hlediska snahy o minimalizaci počtu chyb.

Pozn. 1: Vzhledem ke slučování některých podtémat do zastřešujících témat nelze tento graf brát striktně chronologicky.

Pozn. 2: Zkušenosti se týkají především jednotlivců s vyšší mírou zodpovědnosti, znalosti se týkají týmů jako celku (předpokládá se předávání znalostí zkušenějších jednotlivců méně zkušeným).



5 ZÁVĚR

Význam kvality softwaru v posledních desetiletích neustále vzrůstá. Celosvětová konkurence v globálních ekonomikách přinesla zvýšený tlak na kvalitu, snižování nákladů, rychlost a flexibilitu a zvyšování kvality je v současných podmínkách klíčovou podmínkou úspěchu softwarových produktů. Kvalitní software je výsledkem komplexního přístupu k softwarovému vývoji.

Tato bakalářská práce vznikla jako námět k rozpracování stávajících statistik softwarových chyb zadavatelské firmy (zaměstnavatele). Cílem bylo zpracovat přehlednou typologii softwarových chyb, analyzovat příčiny a faktory vedoucí ke vzniku softwarových chyb a provést srovnávací studii různých postupů pro nalézání příčin chyb v programech s ohledem na potřeby zadavatelské firmy.

Bakalářská práce je poněkud rozsáhlejší, než je vyžadováno; v průběhu zpracovávání projevil zaměstnavatel zájem o detailnější rozpracování kapitol týkajících se postupů odhalování softwarových chyb, zejména ve vztahu k příčinám popsaným ve střední části práce. Koncentrované a logicky zpracované informace budou podkladem ke zvyšování kvality v zadavatelské firmě.

SEZNAM POUŽITÉ LITERATURY

[1] Wikipedia. Software quality [online]. Poslední úprava 11.2.2013 [Cit. 13.2.2013]. Dostupné z: <http://en.wikipedia.org/wiki/Software_quality>.

[2] Smith, D.J.; Wood, K.B. Engineering Quality Software. 2nd Edition. London and New York: Elsevier Applied Science, 1989. 283 s. ISBN 1-85166-358-4.

[3] Guckenheimer, S.; Perey, J.J. Efektivní softwarové projekty. 1. vyd. Brno: Zoner Press, 2007. 255 s. ISBN 978-80-86815-62-6.

[4] ČSN EN ISO 9000: 2006. Systémy managementu kvality - Základní principy a slovník. Praha: Český normalizační institut, 2006. 64 s.

[5] ČSN ISO/IEC 9126-1: 2002. Softwarové inženýrství - Jakost produktu - Část 1: Model jakosti. Praha: Český normalizační institut, 2002. 28 s.

[6] ISO/IEC FDIS 25010: 2010. Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models. ANSI, 2010. 44 s.

[7] Vaníček, Jiří. < vanicek@pef.czu.cz> Kvalita software ve světle mezinárodních norem [online]. [Cit. 2. 3. 2013]. Dostupné z: < http://cev.cemotel.cz/programovani_a_tvorba_sw_1975-2004/2004/311.pdf>.

[8] Patton, R. Testování softwaru. 1. vyd. Praha: Computer Press, 2002. 314 s. ISBN 80-7226-636-5.

[9] Borretzen, J. A; Dyre-Hansen, J. Investigating the Software Fault Profile of Industrial Projects to Determine Process Improvement Areas: An Empirical Study [online]. [Cit. 6. 3. 2013]. Dostupné z: Springerlink. Elektronické databáze VUT.

[10] Eldh, Sigrid; Punnekkat, Sasikumar; Hansson, Hans; Jonsson, Peter. Component Testing Is Not Enough – A Study of Software Faults in Telecom Middleware [online]. [Cit. 6. 3. 2013]. Dostupné z: Springerlink. Elektronické databáze VUT.

[11] Elbaum, Sebastian G.; Munson, John C. Software Evolution and the Code Fault Introduction Process [online]. [Cit. 6. 3. 2013]. Dostupné z: Springerlink. Elektronické databáze VUT.

[12] Mařík, R. Typy softwarových chyb [online]. Publikováno 14. 9. 2007. [Cit. 6. 3. 2013]. Dostupné z: <http://labe.felk.cvut.cz/~marikr/teaching/Y33TSW_10/07.softwaroveChyby.pdf>

[13] Johnston, K.; Page, A.; Rollison, B. Jak testuje software Microsoft. 1. vyd. Praha: Computer Press, 2009. 384 s. ISBN 978-80-251-2869-5.

[14] Walia, G. S.; Carver, J. C. Using error abstraction and classification to improve requirement quality: conclusions from a family of four empirici studies [online]. [Cit. 29. 3. 2013]. Dostupné z: Springerlink. Elektronické databáze VUT.

[15] Eeles, P.; Cripps, P. Architektura softwaru. Nepostradatelný průvodce návrhem softwarové architektury, která funguje. 1. vyd. Brno: Computer Press, 2011. 328 s. ISBN 978-80-251-3036-0.

[16] McConnell, S. Dokonalý kód. Umění programování a techniky tvorby software. 1. vyd. Brno: Computer Press, 2005. 894 s. ISBN 80-251-0849-X.

[17] Couto, C.; Silva, Ch.; Valente, M.T.; Bigonha, R.; Anquetil, N. Uncovering Causal Relationships between Software Metrics and Bugs [online]. [Cit. 3. 4. 2013]. Dostupné z: < http://homepages.dcc.ufmg.br/~mtov/pub/2012_csmr.pdf>.

[18] Martin, R. C. Čistý kód. Návrhové vzory, refaktorování, testování a další techniky agilního programování. 1. vyd. Brno: Computer Press, 2009. 423 s. ISBN 978-80-251-2285-3.

[19] Wikipedia. Technika [online]. Poslední úprava 8.3.2013 [Cit. 23.3.2013]. Dostupné z: <http://cs.wikipedia.org/wiki/Technika>.

[20] Wikipedia. Metodika [online]. Poslední úprava 6.3.2013 [Cit. 23.3.2013]. Dostupné z: <http://cs.wikipedia.org/wiki/Metodika>.

[21] Thapaliyal, M. P.; Verma, G. Software Defects and Object Oriented Metrics. An Empirical Analysis [online]. [Cit. 3. 4. 2013]. Dostupné z: <http://www.ijcaonline.org/volume9/number5/pxc3871859.pdf>.

[22] Sommerville, I. Software Prototyping [online]. Software Engineering, 5th edition, charter 8. [Cit. 23. 4. 2013]. Dostupné z: <https://www.cs.drexel.edu/~bmitchel/course/mcs451/Ch_8.pdf>.

[23] Lacko, B. Testovací prostředky a testovací postupy. In Sborník celostátní konference s mezinárodní účastí Tvorba softwaru 2006. VŠB-TU Ostrava, 2006, str. 102-109. ISBN 80-248-1082- 4.

[24] Lacko,B. Systémový přístup k jakosti softwaru. In Sborník z jubilejního 30.ročníku konference Tvorba software 2004. VŠB-TU Ostrava a Tanger Ostrava, 2004, str. 129-138. ISBN 80-85988-96-8.

[25] Beck, K. Programování řízené testy. Praha: Grada Publishing, 2004. 204 s. ISBN 80-247-0901-5.

[26] Wikipedia. Metodika vývoje softwaru [online]. 4.4.2013 [Cit. 6.4.2013]. Dostupné z: <http://cs.wikipedia.org/wiki/Metodika_v%C3%BDvoje_softwaru>.

[27] Mujumdar, A.; Gayatr M.; Chawan, P. M. Analysis of various Software Process Models [online]. [Cit. 17. 4. 2013]. Dostupné z: <http://www.ijera.com/papers/Vol2_issue3/MA2320152021.pdf>.

[28] Munassar, N. M.A.; Govardhan, A. A Comparison Between Five Models Of Software Engineering [online]. [Cit. 17. 4. 2013]. Dostupné z: < http://www.ijcsi.org/papers/7-5-94-101.pdf>.

[29] Lacko, B. Nové pohledy na životní cyklus tvorby softwaru z hlediska jakosti aplikací automatického řízení. [online]. [Cit. 15. 4. 2013]. Dostupné z: <formular-ekf.vsb.cz/formulare/f01/tsw/getfile.php?prispevekid=758>.

[30] Association of Modern Technologies Professionals. Software Development Methodologies [online]. [Cit. 18. 4. 2013]. Dostupné z: < http://www.itinfo.am/eng/software-development-methodologies/#chapter2>.

[31] Wikipedia. CMMI [online]. Poslední úprava 19. 3. 2013 [Cit. 20. 4. 2013]. Dostupné z: <http://cs.wikipedia.org/wiki/CMMI>.

[32] Wikipedia. Sedm základních nástrojů zlepšování kvality [online]. Poslední úprava 4. 4. 2013 [Cit. 20.4.2013]. Dostupné z: <http://cs.wikipedia.org/wiki/Sedm_z%C3%A1kladn%C3%ADch_n%C3%A1stroj%C5%AF_zlep%C5%A1ov%C3%A1n%C3%AD_kvality>.

[33] Porter, A.; Votta, L. Comparing Detection Methods For Software Requirements Inspections: A Replication Using Professional Subjects [online]. [Cit. 26. 4. 2013]. Dostupné z: Springerlink. Elektronické databáze VUT.

[34] Runeson, P.; Andersson, C.; Thelin, T.; Andrews, A.; Berling, T. What Do We Know about Defect Detection Methods? [online]. [Cit. 28. 4. 2013]. Dostupné z: <http://www.idi.ntnu.no/grupper/su/publ/ebse/X05-know-defectdetectmeth-runeson-ieeeswmay06.pdf>.

[35] Kandt, R. K. A Software Defect Detection Methodology [online]. [Cit. 28. 4. 2013]. Dostupné z: < http://trs-new.jpl.nasa.gov/dspace/bitstream/2014/7184/1/03-0844.pdf>.

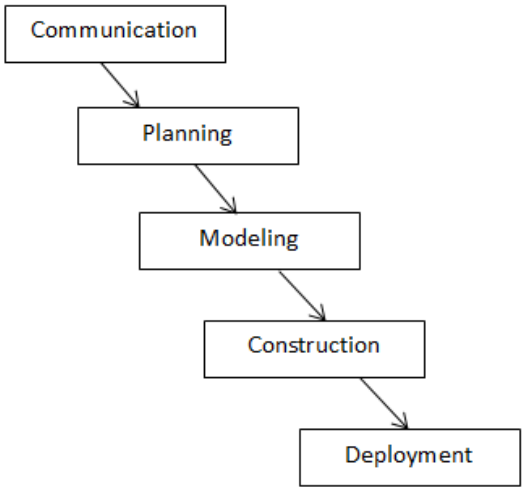
[36] Marri, K. K. Models for evaluating review effectiveness [online]. [Cit. 30. 4. 2013]. Dostupné z: <http://www.infosys.com/IT-services/independent-validation-testing-services/white-papers/Documents/models-evaluating-review.pdf>.

[37] Alexandrov, I. a kol. Impact of Software Review and Inspection [online]. [Cit. 15. 4. 2013]. Dostupné z: <http://hal.in2p3.fr/docs/00/01/34/58/PDF/democrite-00009850.pdf>.

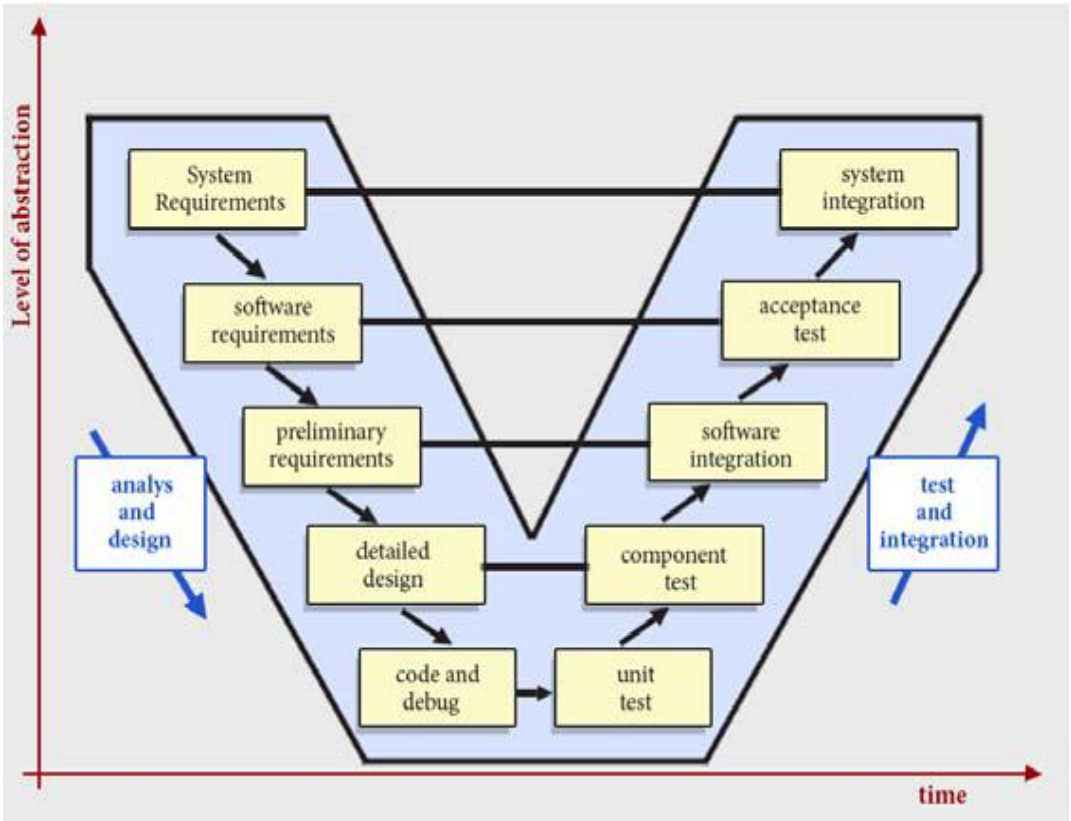
SEZNAM PŘÍLOH

PŘÍLOHY

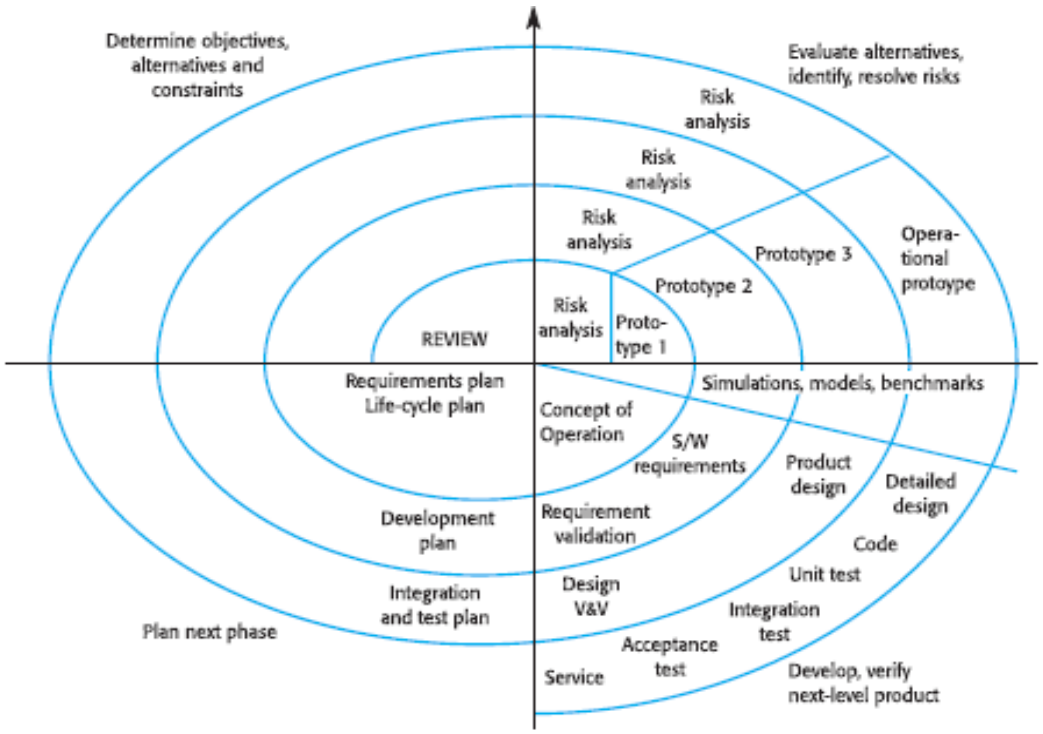
Příloha č. 1. Vodopádový model. [27]



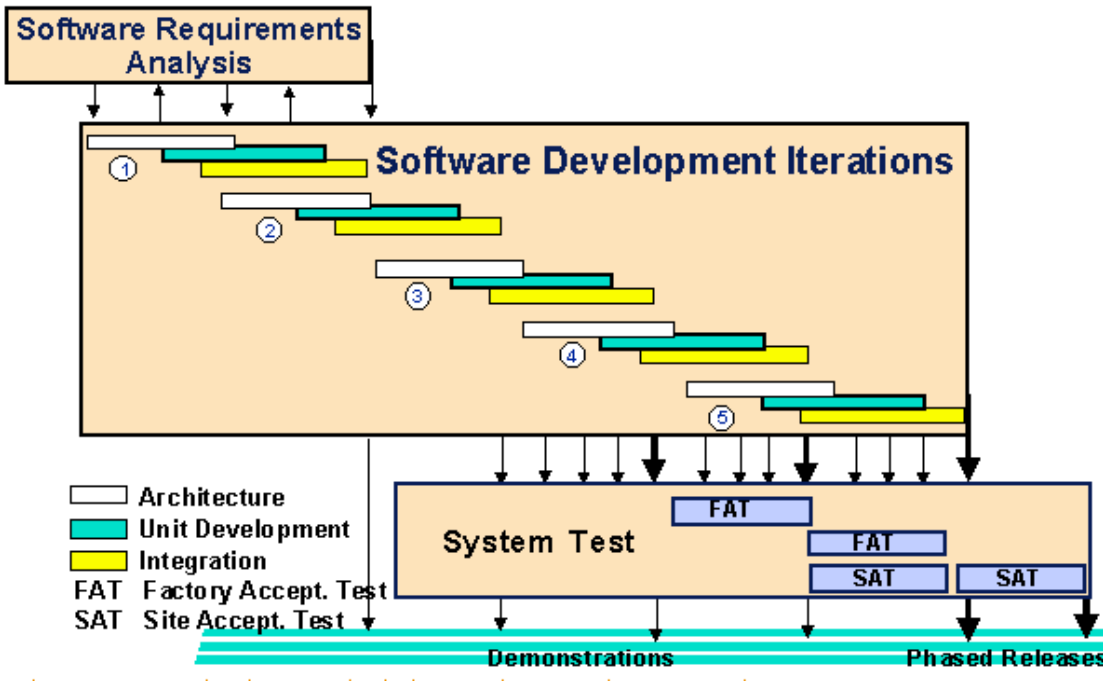
Příloha č. 2. V-model. [28]



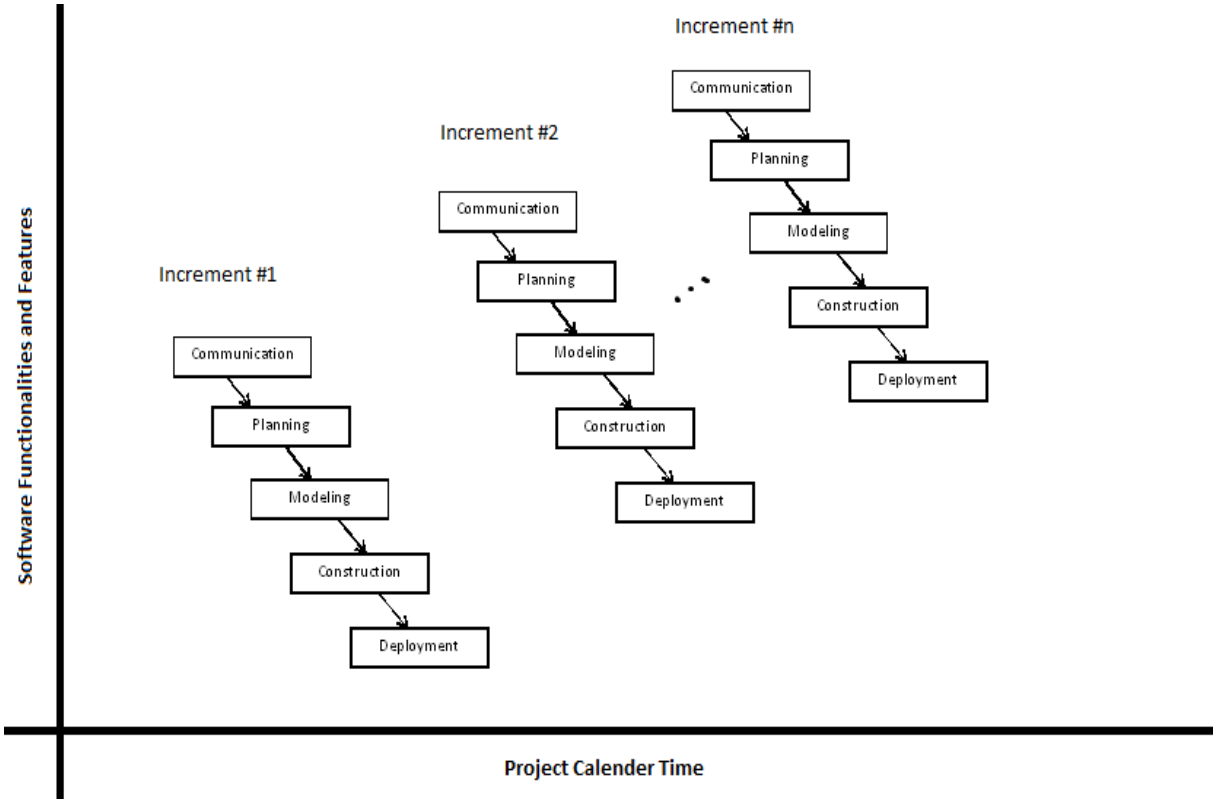
Příloha č. 3. Spirálový model. [28]



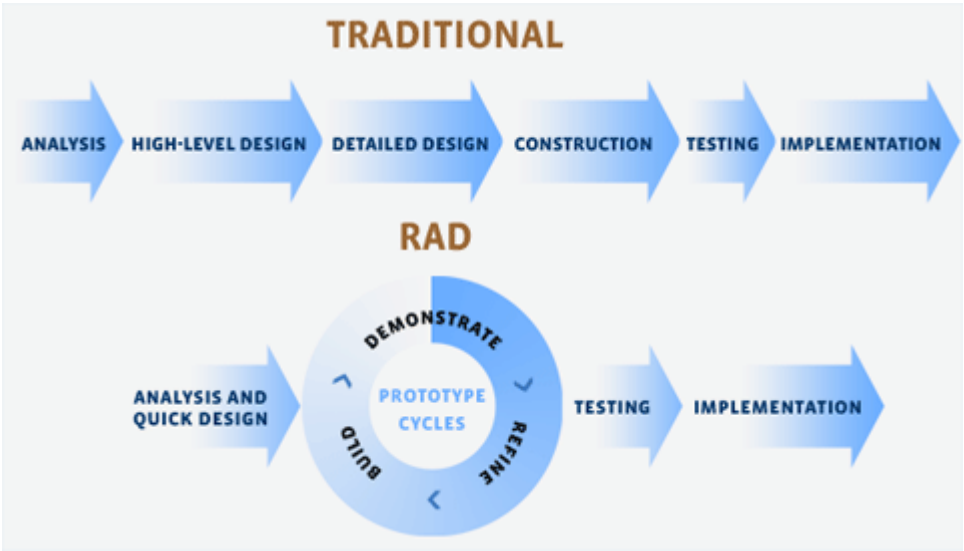
Příloha č. 4. Iterativní model. [28]



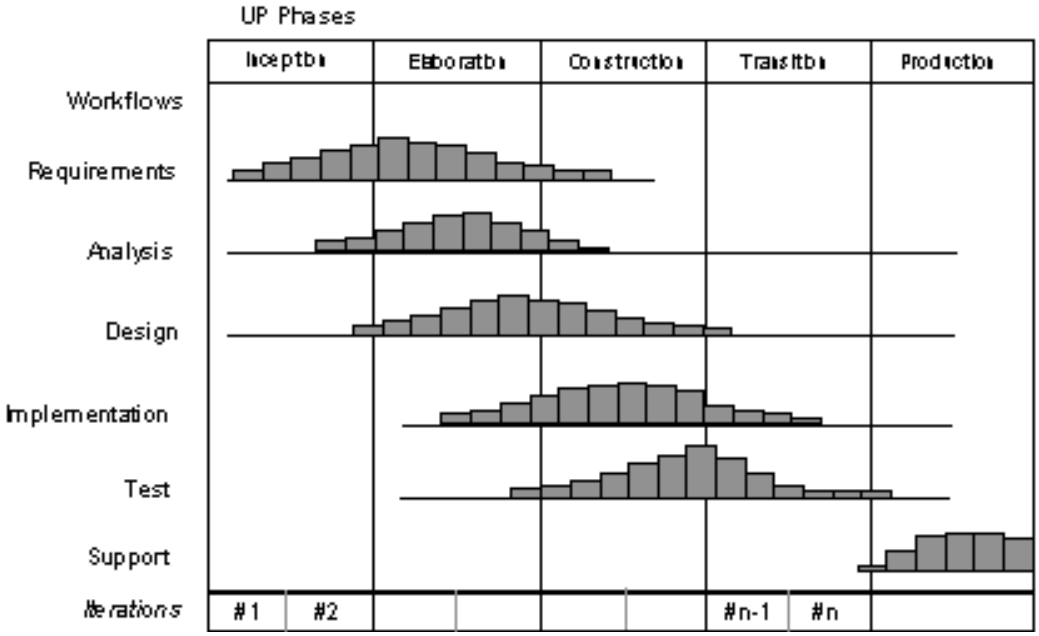
Příloha č. 5. Inkrementální model. [27]



Příloha č. 6. Rapid Application Development.



Příloha č. 7. Rational Unified Process. [27]



Příloha č. 8. Extrémní programování. [28]